GRUMMP Version 0.5.0 User's Guide

Carl Ollivier-Gooch
Department of Mechanical Engineering
University of British Columbia

Copyright © 2010

Carl Ollivier-Gooch

Contents

I Getting and Installing GRUMMP **1**

1 Introduction **2**

1.1 What is GRUMMP? . . . 2
1.2 Goals of the GRUMMP Project . . . 2
1.3 Current Status of GRUMMP . . . 3
1.4 Major Unimplemented Features . . . 4
1.5 Getting Started . . . 4
1.6 Shortcut . . . 4
1.7 Overview . . . 5
1.7.1 Getting GRUMMP . . . 5
1.7.2 Building GRUMMP the easy way . . . 5
1.7.3 Building GRUMMP under Windows . . . 6
1.7.4 Configuring GRUMMP by hand . . . 6
1.7.5 Setting `LD_LIBRARY_PATH` . . . 7
1.7.6 If Your Machine Is Not Yet Officially Supported . . . 8
1.7.7 If Something Goes Wrong During Setup . . . 8
1.7.8 Checking to be Sure Things Built Properly . . . 9
1.8 License . . . 9
1.9 User Services . . . 11

2 GRUMMP Programs **12**

2.1 Common command line options for mesh manipulation executables . . . 12
2.2 Additional options for 2D GRUMMP executables . . . 15
2.3 Additional options for 3D GRUMMP executables . . . 15
2.4 `scat2d` and `scat3d` . . . 16
2.5 `t2d` and `t3d` . . . 17

3 File Formats **18**

3.1 Two-dimensional geometry (`.bdry`) files 18

3.2 Input and output mesh file formats 22

3.3 Three-dimensional geometry (`.bdry`) files 27

3.4 Three-dimensional surface mesh (`.smesh`) files 29

3.5 Three-dimensional ASCII stereolithography (STL) files 29

3.6 Three-dimensional volume mesh input and output (`.vmesh`) files 30

3.7 Mesh quality (`.qual`) files 30

3.8 Status message (`.msg`) files 30

3.9 Scattered data interpolation files 30

II Algorithms Used in the GRUMMP Libraries **33**

4 Mesh Generation **34**

4.1 Two dimensional meshing 34

4.1.1 Initial discretization 35

4.1.2 Point insertion 35

4.1.3 Length scale modifications 36

4.1.4 Curved boundaries 36

4.2 Three-dimensional meshing 40

4.2.1 Initial tetrahedralization 40

4.2.2 Point insertion 40

5 Mesh Improvement **42**

5.1 Swapping 42

5.2 Smoothing 44

5.3 Default mesh optimization in GRUMMP 45

6 Mesh Coarsening **46**

6.1 Selecting Vertices to Retain in the Coarse Mesh 46

6.1.1 Identification of Apexes and Folds 47

6.1.2 Selection of Points to Include in the Coarse Mesh 48

6.2 Incremental Vertex Deletion 48
6.2.1 Invoking Vertex Deletion 49
6.2.2 Mesh Post-processing 50
6.3 Anisotropic Mesh Coarsening 50
6.3.1 Selection of Points in Pseudo-structured Anisotropic Meshes 50
6.4 Outcomes 52

7 Isotropic Mesh Adaptation **53**

A Changes Between Versions **56**
A.1 From version 0.6.3 to 0.6.4: 56
A.2 From version 0.6.2 to 0.6.3: 56
A.3 From version 0.6.0 to 0.6.1/0.6.2: 57
A.4 From version 0.5.0 to 0.6.0: 57
A.5 From version 0.3.3 to 0.5.0: 57
A.6 From version 0.3.3 to 0.4.0: 58
A.7 From version 0.3.2 to 0.3.3: 58
A.8 From version 0.3.1 to 0.3.2: 59
A.9 From version 0.3.0 to 0.3.1: 59
A.10 From version 0.2.1 to 0.3.0: 59
A.11 From version 0.2.0. to 0.2.1: 60
A.12 From version 0.1.7 to 0.2.0: 61
A.13 From version 0.1.6a to 0.1.7: 62
A.14 From version 0.1.6 to 0.1.6a: 63
A.15 From version 0.1.5 to 0.1.6: 63
A.16 From version 0.1.4 to 0.1.5: 64
A.17 From version 0.1.3 to 0.1.4: 64
A.18 From version 0.1.2 to 0.1.3: 65
A.19 From version 0.1.1 to 0.1.2: 65
A.20 From version 0.1.0 to 0.1.1: 66

List of Figures

3.1 Sample 2D boundary data file and geometry. 21
3.2 Orientation of vertices and cells as assumed by GRUMMP I/O routines. 24
3.3 Orientation of boundary data in two dimensions. 25
3.4 Example template file for 2D I/O. 26
3.5 Orientation of cells and vertices for a three-dimensional face. 27
3.6 Sample 3D boundary data file. 28
3.7 Example quality output file 31

4.1 Comparison between a diametral circle (dashed) and diametral lenses. Diametral lenses allow points to be inserted closer
4.2 Framework used for the implementation of generic boundaries 37
4.3 Arbitrary original discretization of a spline. No vertex should be inserted in the shaded areas. 38
4.4 Procedure to follow to recover boundary edges 39

5.1 Face swapping in three dimensions 43
5.2 Edge swapping example 43
5.3 Canonical configurations for edge swapping, including repeat count. 44
5.4 Cross-section of the objective function in an optimization-based mesh-smoothing problem . 45

6.1 Examples of apexes and folds. 47
6.2 Vertex removal by edge contraction in two dimensions 49
6.3 Pseudo-structured surface mesh on a wedge. 51

Part I

Getting and Installing GRUMMP

Chapter 1

Introduction

1.1 What is GRUMMP?

GRUMMP stands for Generation and Refinement of Unstructured Mixed-Element Meshes in Parallel, although it does not yet live up to its acronym. GRUMMP is a set of libraries, written in C++, supporting unstructured mesh creation and modification, and a set of executables built on those libraries.

1.2 Goals of the GRUMMP Project

The goal of the GRUMMP project is to develop automatic mesh generation software for unstructured meshes with mixed element types. The software should produce high-quality meshes that meet user-defined mesh density requirements, using elements appropriate for the geometry and physics of a particular problem.

Automatic mesh generation for complex two and three dimensional domains is a topic of intensive research. It is imperative that automatic mesh generation tools be capable of generating quality finite element and finite volume meshes. There must be a balance between resolution of the boundary and surface features and complexity of the problem. In addition, for problems with isotropic physics, element aspect ratio must be small to minimize linear system condition number and interpolation error. On the other hand, problems with anisotropic physics (for example, a shear layer in viscous fluid flow) require highly anisotropic elements for efficient solution. A further level of complication is that for some physical problems and applications, quadrilateral (2D) or hexahedral (3D) elements are preferred, even though filling space with high quality elements is easier using triangular (2D) or tetrahedral (3D) elements.

A general-purpose automatic mesh generator should address all of these issues without excessive user intervention. We envision a system in which common types of physical problems have predefined mesh sizing and element aspect ratio functions, allowing easy generation of meshes for these applications areas. For flexibility and generality, the user will also be able to prescribe these functions (for totally different applications) or modify the predefined behaviors (to provide a quality mesh in the wake of an airplane wing, for example).

GRUMMP addresses these issues by implementing mesh manipulation primitives to generate of modify existing meshes so that criteria for element size and quality are met. In addition, automatic computation of local length scale is performed to provide a default in cases where solution-based adaptive length scales are not available.

1.3 Current Status of GRUMMP

The current release of GRUMMP is version 0.6.4.[1] This version consists of eight executable programs.

- `tri` generates two-dimensional triangular isotropic unstructured meshes, using a seriously enhanced version of Ruppert's algorithm. Recent work at UBC has extended Ruppert's scheme to allow better user control over cell size and grading [11] and to allow meshing with curved boundaries [1].
- `meshopt2d` improves existing two-dimensional triangular isotropic unstructured meshes, including performing adaptive refinement based on a user-provided length scale.
- `coarsen2d` produces a triangular unstructured mesh with approximately twice the local length scale of the input mesh, including directional anisotropic coarsening of quasi-structured parts of the fine mesh.
- `tetra` generates three-dimensional tetrahedral isotropic unstructured meshes, using Shewchuk's algorithm [16] modified for cell size and grading control [11].
- `meshopt3d` improves existing three-dimensional tetrahedral isotropic unstructured meshes, using techniques developed by Freitag and Ollivier-Gooch [3]. Just as in 2D, this executable also performs adaptive refinement.
- `coarsen3d` produces a tetrahedral unstructured mesh with approximately twice the local length scale of the input mesh, including directional anisotropic coarsening of quasi-structured parts of the fine mesh.
- `scat2d` performs linear interpolation of two-dimensional scattered data.
- `scat3d` performs linear interpolation of three-dimensional scattered data.

For further information on invoking these programs, including a description of command line arguments, see Chapter 2. For a description of the file formats which GRUMMP executables read and write, see Chapter 3.

[1] Versions 0.6.x are intended to be a fully-stable and well-supported set of executables. Because GRUMMP is distributed in source form, the libraries that these executables call are in principle accessible for programming use. While this is permitted by the terms of the GRUMMP license (see Section 1.8), the library interface is neither documented nor supported at this time. Until the GRUMMP API is officially publicly released, interested users can also work with the GRUMMP libraries through the ITAPS mesh interface (`http://itaps-scidac.org`), which currently supports mesh query and low-level mesh modification operations.

1.4 Major Unimplemented Features

These are listed approximately in order by priority, along with estimates of when each feature is likely to be completed, if known.

- Mesh anisotropy. There are two (possibly) related issues here: anisotropy meshing and anisotropic mesh refinement.
  - Anisotropic refinement, beginning with anisotropic tri/tet meshes and moving eventually to anisotropic mixed-element meshes. (2009 thesis completion for 2D; 3D currently in progress)
  - Mixed-element meshing, including quadrilaterals in two dimensions and pyramids, prisms, and hexahedra in three dimensions. (2D: possibly as early as 2011) Ideally, we'd like to be able to produce mixed-element meshes after anisotropic refinement as well.
- Parallelism. We are currently working to parallelize GRUMMP, including both its internal operations and an API for parallel access to data structures (this is part of the ITAPS project).

1.5 Getting Started

1.6 Shortcut

The simplest way to build GRUMMP is this:

1. Download the latest tarball from ftp://tetra.mech.ubc.ca/pub/GRUMMP/GRUMMP.tar.gz. Anonymous ftp, wget, and your web browser should all work here.
2. Unpack the tar ball:

   ```
   tar -zxvf GRUMMP.tar.gz
   ```

3. Go to the directory that tar just created (this includes a version number; the tar output will show you what to use).
4. Run `configure`.
   (a) If you have CGM installed, tell configure where it is using `--with-CGM-path=/path/to/CGM`. If you don't have CGM installed, GRUMMP will do its best to download and configure it at this point[2], then build it when you run `make` later. This auto-build may fail for some systems where downloads require special magic; if you hit this, then download and build CGM yourself first.
   (b) To enable support for the ITAPS mesh interface, specify `--enable-itaps`
   (c) Other standard configure options apply, as well as some GRUMMP-specific ones. Use `configure --help` or read the rest of this document for more information.
5. Run `make`, then `make test` to confirm success.
6. To install the libraries and executables, run `make install`.

[2]Fine print: `wget` will be used to download the tarball from `http://www.mcs.anl.gov/~tautges/downloads/CGMA-latest.tar.gz`.

1.7 Overview

GRUMMP is distributed in source form. The software is self-configuring, using a script generated by GNU `autoconf`. Note that `autoconf` is *not* required to set up GRUMMP. The primary development platform for GRUMMP is Linux, with gcc4; unfortunately, we do not currently have access to a wide range of platforms for thorough portability testing. However, other Unix variants should pose no major problems, especially with GNU compilers. In addition, native compilers on the following platforms have worked in recent releases and so are expected to still work:

- x86 / Linux 2.x with Intel v5.0 compilers
- SGI / IRIX 6.x
- Compaq-Alpha-HP / OSF 5.1 with Compaq compilers (tested on a Compaq-branded machine, but they all use Compaq's compiler, so all should be fine, as should Alpha / Linux machines with the Compaq compiler)
- IBM / AIX with xlC
- x86 / Win32 with VC++
- x86 / Win32 + Cygwin with gcc. Support here isn't perfect; please report any problems. In particular, it isn't at all clear that shared libraries will work on Cygwin at this point.

On some systems, warnings occur during compilation; none of these warnings are thought to affect executable performance. Other systems may also work, but have not been tested in some time due to lack of access to test machines.

1.7.1 Getting GRUMMP

GRUMMP is most easily available on the WWW from the GRUMMP home page `http://tetra.mech.ubc.ca`. GRUMMP version 0.6.4 `ftp://tetra.mech.ubc.ca/pub/GRUMMP/GRUMMP-0.6.4.tar.gz` is also available via anonymous ftp. Or you can choose the symbolic link to the most recent version `ftp://tetra.mech.ubc.ca/pub/GRUMMP/GRUMMP.tar.gz`.

1.7.2 Building GRUMMP the easy way

Suppose that you download GRUMMP and store the gzipped tar file in /home/me. After using `gunzip` and `tar` to extract the files from the distribution,

```
cd /home/me/GRUMMP-0.6.4
```

Now all you need to do is type

```
./GRUMMP-build
```

This runs a shell script that will automatically configure and build GRUMMP for your machine, if at all possible. The script first attempts to build an optimized version of GRUMMP. If that fails, then it tries again without optimization (and with more compiler warnings enabled). Any compiler error and warning messages are stored in a temporary file. After the script is finished building GRUMMP, it summarizes the outcome for you.

Finally, the script will collect relevant output files and create a tar file of them. Email-ing this file to the developers at `cfog@mech.ubc.ca` will help us to improve portability of GRUMMP. The tar file contains only output files from configuring and building GRUMMP.

1.7.3 Building GRUMMP under Windows

There is a VC++ project file available in the top-level directory of the GRUMMP distribution. In principle, all you should have to do is use this project file. Minor problems have been reported, however. If you should happen to run into problems (especially if you have fixes for them to pass along), please let the developers know, as we don't do our own Windows testing in-house.

1.7.4 Configuring GRUMMP by hand

Under some circumstances, GRUMMP may not build properly automatically for reasons unrelated to the source code. The most common reasons are the `flex`/`lex` libraries are in an odd place, or that the native compiler on your machine has an unusual name. In these cases, running `configure` yourself will likely solve the problem.

Suppose that you download GRUMMP and store the gzipped tar file in /home/me. After using `gunzip` and `tar` to extract the files from the distribution,

```
cd /home/me/GRUMMP-0.6.4
```

and type

```
./configure
```

The `configure` shell script is an automated process for determining the type of machine that GRUMMP will be running on and the capabilities of the system software on the machine.

`configure` options

`--with-flex-lib-dir=DIRNAME` This option is sometimes needed so that `configure` can find the library `libfl.a` (`flex`) or `libl.a` (`lex`).[3] This option is not needed unless `configure` exits with a warning about not being able to find these libraries *and* you want to use user-defined file formats (see Chapter 3). In this case, re-run `configure` with the full path for the library using this option.

[3] `flex` is used if found; otherwise `lex` is used. If neither is found, then user-defined I/O formats can not be used.

--with-c-compiler=NAME

--with-cxx-compiler=NAME Specify the compiler name using these options to use a C/C++ compiler other than the native compiler (including `gcc`/`g++`), to use a native compiler that is in an unusual place, or to use `cc`/`CC` on Sun machines (older Sun machines have a broken native C compiler). Only the name of the compiler is needed if it resides in your usual path. Full path names are accepted, as are relative path names from directories in your default path.[4] If in doubt, use the full path name. In any event, output from `configure` will tell you which compiler will be used.

--with-addl-path=PATH Specify an additional entry for the `PATH` environment variable. This helps configure find its way to compilers located in odd places. I don't fully understand why this is even necessary, but I had to add it to compile version 0.1.7 on one of my test machines.

--with-debug This option enables careful compile-time checking of the code and run-time verification that is much more extensive than in the optimized version, as well as producing an executable that can be used in conjunction with a debugger. This is primarily a development option, but end-users are occasionally asked to use this option in conjunction with bug reports that can not be reproduced on machines to which the developers have ready access.

--enable-itaps This option builds GRUMMP with support for the iMesh interface developed by the Interoperable Tools for Advanced Petascale Simulations (ITAPS) consortium. This API provides low-level mesh query and modification functionality in a data-structure and programming language independent way. See `http://itaps-scidac.org` for more information.

–with-CGM-path This specifies the location of a recent version of the Argonne Common Geometry Module. If this option is omitted from the configure command, configure will automatically download and build CGM.

Once `configure` has finished, type

```
make -k
```

This nested `make` will create all the libraries required by GRUMMP as well as the executables.

1.7.5 Setting `LD_LIBRARY_PATH`

GRUMMP builds both shared and static libraries. By default, the shared libraries are used to create the executables. To be able to run these executables, you must add `/home/me/GRUMMP-0.6.4/lib/lib[gO]/`*`system-type`* to the environment variable `LD_LIBRARY_PATH`. *`system-type`* is a variant on the name of your OS. `GRUMMP-build` and `configure` print this information at the end of its run, including information on how to change the environment variable. If in doubt, check to see what directory was just created in `GRUMMP/lib`.

[4] For example, `/usr/local/lib/cc` could be specified as `../lib/cc` if `/usr/local/bin` is in your default path. Note that `/usr/lib/cc` would be found instead in this case if `/usr/bin` precedes `/usr/local/bin` in your path.

1.7.6 If Your Machine Is Not Yet Officially Supported

First, go ahead and try to configure and build GRUMMP anyway, with this minor variation: use

```
./configure --with-debug [other config options] >& config-output
```

This will enable at least some compile-time checking as well as enabling debugging. Then build GRUMMP using (with `csh`-type shells)

```
make -k >& make-output
```

The last few lines of `make-output` will tell you whether the build was successful. Whether the build is successful or not, please mail the following information to `cfog@mech.ubc.ca`:

- `config.status`
- `config.log`
- `config.cache`
- `include/GR_config.h`
- `make-output`
- `config-output`
- result of `uname -a`
- If your compiler is not `gcc/g++`, please also include the parts of your C and C++ compiler man pages that describe optimization, debugging, and enabling warnings.

This information will be useful in developing full support for new systems. In particular, `make-output` and the information on enabling warnings can be used to improve portability; and information on compiler optimization flags will improve out-of-the-box optimization of GRUMMP.

1.7.7 If Something Goes Wrong During Setup

Problems with `configure`

If `configure` fails, it will exit with an error message. Please report this message to `cfog@mech.ubc.ca`, along with copies of the files `config.status`, `config.cache`, and `config.log`, if they exist, and the result of `uname -a` on your machine. If more information is needed to diagnose a misconfiguration problem, you will be contacted. `configure` failures are quite rare: I can only recall one so far.

Problems with `make`

If `make` fails, please re-run `configure` and `make` as described in Section 1.7.6 and submit the same information requested there. At least some machines may require additional configuration support for certain system libraries, and this information will help isolate these problems.

1.7.8 Checking to be Sure Things Built Properly

In the GRUMMP root directory, run

```
make test
```

This will use the newly-created GRUMMP executables to generate several meshes in both 2D and 3D, as well as improving a sample mesh in 3D. If all has gone well, the only output from this command should be:

```
Testing 2D mesh generation...success
Testing 3D mesh generation...success
Testing 3D mesh improvement...success
```

Should you get a failure here, please report it to the developers.

1.8 License

GRUMMP 0.6 License Agreement

The Software included in the GRUMMP 0.6 distribution is not in the public domain. It is Copyright 1997-2001 by The University of British Columbia and The University of Chicago. As of 2001, copyright was transfered to the principle author, Carl Ollivier-Gooch. However, GRUMMP is available for license, without fee, for educational and non-profit research purposes. Contact information for the developer:

Carl Ollivier-Gooch
Dept of Mechanical Engineering
The University of British Columbia
2324 Main Mall
Vancouver, BC V6T 1Z4
Canada

Fax: (604)822-2403
Email: cfog@mech.ubc.ca

1. **Definitions.** The "**Software**" refers to all parts of the GRUMMP 0.6 distribution, in source code, object code, or executable code form. The "**Library**" refers to linkable object-code libraries created from the source code. The "**Executables**" refers to the included programs for 2D/3D mesh generation (`tri`, `tetra`), for 2D/3D mesh quality assessment and improvement (`meshopt2d`, `meshopt3d`), 2D/3D mesh coarsening (`coarsen3d`,`coarsen3d`) and 2D/3D scattered data interpolation (`scat2d`, `scat3d`). A "**work using the Library**" means a work that links to the Library but does not include any of the source code of the Library (except header files) or modify the Library in any way. A "**work based on the Software**" means a work containing all or a portion of the source code, either verbatim or with modifications, or any other derivative of the Software under copyright law. Each licensee is addressed as "**you**" or "**Licensee**".

2. **Copyright holders.** The Software is copyrighted, and copyright is owned by Carl Ollivier-Gooch (the "**Copyright Owner**"). The Copyright Owner reserves all rights except those expressly granted to the Licensee herein and U.S. Government license rights.

3. **Use of the Software.** Licensee shall use the Software solely for education and non-profit research purposes within Licensee's organization. Permission to copy the Software for use within Licensee's organization is hereby granted to Licensee, provided that the copyright notice and this license accompany all such copies. Licensee shall not permit the Software to be used by persons outside Licensee's organization. Licensee shall not have the right to relicense or sell the Software or to transfer or assign the Software.

4. **Works using the Library.** The development and not-for-profit distribution of works using the Library by educational or non-profit research organizations is encouraged. Such distribution requires a separate no-fee license; please contact the Copyright Owner for further information. Development of commercial products using the Library is also encouraged; contact the Copyright Owner to discuss licensing requirements.

5. **Works based on the Software.** The Software can be modified and redistributed within the Licensee's organization provided all copyright notices are left intact and changes in the source code are prominently marked. Redistribution outside the Licensee's organization is prohibited.

6. **Commercial use of the Software.** Commercial concerns interested in using the Software, creating works using the Library, or creating works based on the Software should contact the Copyright Owner for licensing information.

7. **Acknowledgement of use.** Licensee agrees to acknowledge use of the Software in any document referencing work using the Software, including but not limited to published research. Also, licensee agrees to notify The University of British Columbia of any such document and supply a copy of the document to the developer for the University, with appropriate accommodation made for the protection of confidential or proprietary data.

8. NEITHER THE COPYRIGHT OWNER, THE UNIVERSITY OF BRITISH COLUMBIA, THE UNIVERSITY OF CHICAGO, THE UNITED STATES GOVERNMENT NOR ANY OF THEIR EMPLOYEES MAKE ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED AND COVERED BY A LICENSE GRANTED UNDER THIS LICENSE AGREEMENT, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

IN NO EVENT WILL THE COPYRIGHT OWNER, THE UNIVERSITY OF BRITISH COLUMBIA, THE UNIVERSITY OF CHICAGO, OR THE U.S. GOVERNMENT BE LIABLE FOR ANY DAMAGES, INCLUDING DIRECT, INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM EXERCISE OF THIS LICENSE AGREEMENT OR THE USE OF THE LICENSED SOFTWARE.

9. Portions of the Software resulted from work under a U.S. Government contract and are subject to the following license: the Government is granted for itself and others acting on its behalf a paid-up nonexclusive, irrevocable worldwide license in this computer software to reproduce, prepare derivative works, and perform publicly and display publicly.

1.9 User Services

Bug Reporting Report bugs to `cfog@mech.ubc.ca`. Always include the command line used and the error message that was given. If at all possible, include the input file (compressed, for large files) as well to be sure that I can reproduce the problem.

WWW Site The official web site for GRUMMP is `http://tetra.mech.ubc.ca/GRUMMP`.

Chapter 2

GRUMMP Programs

GRUMMP includes executables for two- and three-dimensional mesh generation (`tri` and `tetra`), mesh improvement (`meshopt2d` and `meshopt3d`), mesh coarsening (`coarsen2d` and `coarsen3d`), and interpolation of scattered data (`scat2d` and `scat3d`) (see Section 2.4). The only major difference among the meshing executables is their starting data:

- `tri` and `tetra` begin by creating a valid initial mesh based on the boundary geometry.
- `meshopt2d` and `meshopt3d` read a valid mesh.
- `coarsen2d` and coarsen3d read a valid mesh and initially coarsen the mesh by reducing the number of vertices by a factor of approximately 2^D.

From this point, all these executables combine point insertion, point deletion, local mesh reconnection, and local smoothing algorithms to generate high-quality meshes.Not surprisingly, these executables share a number of command-line options. These common options are described in Section 2.1; additional options are described in Sections 2.2 and 2.3. All of these executables generate mesh quality files (see Section 3.7). In addition to the progress and status messages displayed on the screen, each executable also saves a more copious set of messages in a file (see Section 3.8).

2.1 Common command line options for mesh manipulation executables

All of the GRUMMP mesh generation, improvement, and coarsening executables share a number of command-line options in common.

-i *`basefilename`* This argument is mandatory and specifies the stem of the mesh file name; both relative and absolute path names are acceptable. Extensions will be added to this stem as appropriate for mesh input and output, quality file output (`.qual`), and message output (`.msg`). For example,

the message output file will be *`basefilename`*`.msg`. Because *`basefilename`* is used in its entirety, all output files will be in the same directory as the input file.

For `tri` and `tetra`, the default input file extension is `.bdry`; these files specify the boundary of the domain to be meshed in terms of its underlying geometry. See Sections 3.1 and 3.3 for information on the format of these files.
The output file format and extensions for all the GRUMMP meshing executables are user-definable. In particular, it is possible to write node locations in one file and connectivity in another, or any other combination of dividing information between files. This capability allows the programs to exchange data with any other preprocessor or analysis software that uses ASCII files for mesh data. A full description of how to use this feature of GRUMMP can be found in Section 3.2.
The mesh modification executables (`meshopt[23]d` and `coarsen[23]d`) append `.out` to the output file names they would otherwise use to prevent possible file name collisions with input files.
In addition to writing output in a wide range of file formats, `meshopt2d` and `coarsen2d` can read any file format that they can write.[1]

-A `arg` If the argument is non-zero, use Jonathan Shewchuk's adaptive geometric predicates to evaluate orientation and in-ball primitives. These predicates are carefully designed to be extremely efficient, so there is not a large performance penalty for using them. On the other hand, my experience is that there is little benefit in terms of robustness to using these predicates. There may, however, be cases where this helps, so feel free to try this option in difficult cases. **Default: 0.**

-b `arg` If the argument is non-zero, the boundary of the mesh is treated as precious: no points are inserted on the boundary, boundary points are not moved by smoothing, and local reconnection in three dimensions does not change boundary connectivity. Notwithstanding, `tetra` will insert points on the boundary if necessary to create a constrained tetrahedralization. *Use of this argument is not recommended, because it can severely degrade mesh quality!* **Default: boundary is changeable.**

-q `arg` Specify a cell quality measure to be evaluated; the quality data is binned for use in producing histograms; the maximum and minimum quality measures are also retained for each evaluation. Five quality measures are supported in two dimensions, and ten in three dimensions. Note that this option affects output only: no effort is made to optimize using the selected measure.
For each measure, the accompanying table shows the minimum and maximum attainable values, along with the number of bins and range which those bins cover; the latter is more than the attainable range for angle measures. **Default: 2 (all angles [dihedrals in 3D])**.

Maximum/minimum/all angles in the cell (both solid and dihedral angles are available in three dimensions).

Marcum's sliver measure [3D]. This measure takes the ratio of tetrahedron volume to the cube of the mean edge length, with normalization so that an equilateral tetrahedron has quality 1. The measure is designed to have low values for *slivers*, tetrahedra which have both very large and very small dihedral angles.

[1] In fact, the input and output formats need not be the same, so it is possible to simultaneously improve a mesh and change its file format.

Marcum's skewness measure. [3D] This measure takes the ratio of volume to the cube of the circumradius of a tetrahedron; again, normalization makes quality equal to 1 for an equilateral tetrahedron. This measure is designed to detect skewed tetrahedra. For more information on these last two quality measures, see [4].

Aspect ratio I, the ratio of inscribed circle (sphere) radius to circumcircle (sphere) radius, with a constant multiplier included so that an equilateral cell has quality 1. A degenerate cell has quality 0 according to this measure.

Aspect ratio II, the ratio of area of the triangle (volume of tetrahedron) to perimeter squared (surface area to the $3/2$). Again a constant multiplier is used so that an equilateral cell has quality 1 and a degenerate cell has quality 0.

Shortest edge to circumradius, the ratio of the shortest edge length to the circumradius of the cell. This is the quality measure that Jonathan Shewchuk's scheme guarantees to produce excellent values for, and is used during 3D insertion. A shortcoming of this measure is that 3D slivers, which are bad cells, can have high values of short-edge to circumradius ratio. This measure is normalized so that a perfect cell has quality of 1 and most degenerate cells have quality 0.

Measure	`arg`	Min value	Max value	Bins	Min bin	Max bin
Max angle (2D)	0	60	180	30	0	180
Min angle (2D)	1	0	60	30	0	180
All angles (2D)	2	0	180	30	0	180
Max dihed (3D)	0	70.5	180	30	0	180
Min dihed (3D)	1	0	70.5	30	0	180
All diheds (3D)	2	0	180	30	0	180
Max solid (3D)	3	31.5	360	30	0	360
Min solid (3D)	4	0	31.5	16	0	32
All solids (3D)	5	0	360	30	0	360
Sliver (3D)	7	0	1	25	0	1
Skewness (3D)	8	0	1	25	0	1
AR I	6	0	1	25	0	1
AR II	9	0	1	25	0	1
Short edge to radius	10	0	1	25	0	1

-g *G* Change the rate at which cell size can change as you move across the mesh. Length scale in the mesh can change by no more than d/G as you move a distance d. Accordingly, larger values of G lead to mesh that are more uniform. **Default: 1.**

-r *R* Resolve geometric features with about R cells. Increasing R therefore increases both resolution and mesh size. This option replaces the old -s option.

-l *filename* (That's the letter ell, not the number 1.) This option can be used to specify length scale for mesh refinement: that is, adaptive refinement (`meshopt2d/3d`). Specifically, each line in the file

should contain a vertex index and a length scale for that vertex; some or all vertices can have length scale specified, in any order. This length scale is assigned to the given vertex, and mesh grading updates the length scale for nearby vertices.

2.2 Additional options for 2D GRUMMP executables

There are two command line options that apply only to the 2D GRUMMP executables.

-c `arg` If the argument is non-zero, protect boundary edges from nearby insertion using diametral lenses instead of diametral circles. Diametral lenses are smaller than diametral circles, and result (both in theory and in practice) in better meshes. Robust values are 0 and 1. An experimental extension that gives somewhat better meshes in practice but is less robust is available with the value 2. **Default: 1.**

`-j arg` If the argument is non-zero, coarsen anisotropic meshes with the goal of reducing cell aspect ratio. **Default: 0.**

2.3 Additional options for 3D GRUMMP executables

There are several command line options that apply only to the 3D GRUMMP executables.

-a `arg` If the argument is non-zero, allow changes in the surface mesh shape that do not involve angle of more than `arg` degrees. **Default: 5.**

-e `arg` If the argument is non-zero, allow local reconnection via edge removal. **Default: 1.**

-O `string`

Specify a shorthand mesh optimization script. Although the automatic mesh improvement scheme creates excellent meshes, it can be quite slow. For this reason, the optimization string option has been revived. The string encodes a sequence of actions for optimization. For more information about the individual optimization procedures, see [3]. The full list of supported options is:

Swapping. There are three swapping suboptions: '`wi`', '`wm`', and '`ws`'.

`wi` Insphere swapping. Use of this criterion results in a mesh which is locally Delaunay with respect to all faces which can be swapped either 2 tets for 3 or 3 for 2. The Delaunay tetrahedralization has, in principle, a number of excellent properties. Insphere swapping essentially tries to make tetrahedra as equilateral as possible, but it has blind spots which allow tetrahedra with very small or very large dihedral angles to remain and even proliferate in the mesh. This is often a good choice for a *first* pass of swapping when improving a mesh, but a very poor choice for later passes, in that it will undo much of the good done by other operations. `tetra` produces a Delaunay mesh at the end of its insertion process anyway, so this option is of no use with `tetra`.

`wm` Minmax dihedral angle. This criterion performs swaps if and only if the new configuration has a smaller maximum dihedral angle than the old configuration. This criterion does a better job of eliminating poor angles from the mesh than the insphere criterion, but does not do a spectacular job with small dihedral angles.

`ws` Maxmin sine of dihedral angle. This criterion performs swaps if and only if the new configuration has a larger minimum sine of dihedral angle than the old configuration. As such, both very small and very large dihedral angles are removed. ***This is the recommended swapping criterion.***

Smoothing. A smoothing command (’`mf`’)[2] is followed by a single digit to specify the local mesh quality function to use. The options are:

1 Maxmin dihedral angle.

2 Minmax dihedral angle.

3 Maxmin cosine of dihedral angle.

4 Minmax cosine of dihedral angle.

5 Maxmin sine of dihedral angle. ***Recommended.***

The objective of each option is self-explanatory from the name of the quality measure. In each case, the measure is evaluated; a single Laplacian smoothing step is made; the quality measure is re-evaluated; the Laplacian step is kept if and only if the mesh is improved by it; and a non-smooth optimization procedure is invoked if the local sub-mesh is not yet sufficiently good.

Bad tetrahedron repair. Use ’`r`’ to repair bad tetrahedra using a full range of swapping techniques, and ’`R`’ to additionally use strategic point insertion to remove bad tetrahedra; the latter should be used with caution.

The **default** optimization string for `tetra` is “**`-O wsmf5`**”. For `meshopt3d`, the value recommended by Freitag and Ollivier-Gooch [3] is used: “**-O `wsmf5mf5rmf5mf5`**”. The difference between the two is that `tetra` is known to create meshes that are reasonably good to begin with and need little cleanup, whereas `meshopt3d` can’t make any assumptions about the quality of the input mesh.

2.4 `scat2d` and `scat3d`

`scat2d` and `scat3d` perform interpolation of scattered data. An input data file, containing both locations at which data is known and the data itself, is read. The points are triangulated (as part of a larger Cartesian box); swapping is performed to obtain reasonable connectivity. A tree structure containing the data points is constructed to facilitate interpolation. A second set of points is read in; these are the target locations for the interpolation. Data is interpolated to these points and written out. For file formats, see Section 3.9.

The command line arguments for `scat2d` and `scat3d` are:

[2] There are in fact other options supported by the underlying smoothing code. However, the option invoked by `mf` is so strongly recommended that the other options will not be discussed here.

-i *basefilename*

This argument is mandatory and specifies the stem for all file names used by the program. The input file containing the data to be interpolated is *basefilename*.data; the input file containing locations to which to interpolate the data is *basefilename*.points; the optional output volume mesh file is *basefilename*.vmesh.out; the message file is *basefilename*.msg; and the output file for interpolated data is *basefilename*.out.

-m This optional argument directs scat2d/scat3d to save the 2D/volume mesh it generates in *basefilename*.mesh/*basefilename*.vmesh. **Default: don't save mesh.**

2.5 t2d and t3d

The programs t2d and t3d[3] translate two-dimensional (respectively, three-dimensional) mesh files from one format to another. To use these programs, enter the GRUMMP/src/IO_generator directory. Edit the files t[23]d_in.template and t[23]d_out.template; see Section 3.2 for information on the format of these files. Then type make t[23]d.[4] The resulting executable will read a mesh in the format specified in t[23]d_in.template and write it in the format specified in t[23]d_out.template. The only argument to t[23]d is -i *basefilename*, which has the same meaning as for the other executables. Again, output files have .out appended to their names to avoid filename collisions.

[3]For the rest of this section read t[23]d as t2d for two dimensions or t3d for three dimensions.

[4]You must have a working version of lex or flex on your machine to build t2d or t3d successfully.

Chapter 3

File Formats

3.1 Two-dimensional geometry (`.bdry`) files

Two-dimensional domains are specified using a geometry (`.bdry`) file. This file format supports simultaneous and consistent meshing of multiple sub-domains and is extensible to non-polygonal boundaries, using the boundary patches described below.

The first line of the file contains the following:

```
NPoints NPatches [NPeriodic]
```

i.e. the number of points and the number of boundary patches in the file; the number of pairs of periodic patches is optional. The next **`NPoints`** lines give point coordinates. Note that points will not necessarily be present in the mesh obtained; they are only used to define the boundary patches.

The **`NPatches`** boundary patch descriptions follow the vertex coordinates. The descriptions use the following format:

```
patchname leftsidetype leftnumber rightsidetype rightnumber [extra_info]
```

where **`patchname`** can take the following values (case insensitive):

- **`polyline`**, for polygonal boundaries
- **`circle`**, for circles
- **`arc`**, for circular arcs of less than 180 degrees
- **`longarc`**, for circular arcs of more than degrees
- **`bezier`**, for Bézier curves

- **`spline`**, for cubic interpolated splines

Because GRUMMP supports meshing of multiple subdomains, both boundary condition and subdomain identification must be included in the input, including the relationship of the various subdomains to the boundary patches. If the patch has a subdomain to be meshed on its left side, then **`leftsidetype`** should be **`r`**, and **`leftnumber`** should be a subdomain index between 1 and 126 inclusive. If the patch has a boundary on its left side, then **`leftsidetype`** should be **`b`**, and **`leftnumber`** should be an integer boundary condition tag strictly greater than zero. The same holds for the right side information for each patch.

Note that a patch can have a subdomain on both sides, but not a boundary on both sides. We are aware that the ability to specify boundary conditions on internal boundaries is sometimes necessary (our current favorite example involves radiative heat transfer onto a surface). While there are plans to add this capability in future versions of GRUMMP, there is no simple way to do this at present.

The **`[extra_info]`** varies depending on the boundary patch type. It usually consists of a list of point indices. Note that the point numbering starts at zero. Here is the required format for all possible boundary patches.

Polyline boundary patch

The format for **`[extra_info]`** is:

`numberpoints points`

where **`numberpoints`** is the number of points used to describe the polygonal boundary. Then follow **`numberpoints`** indices to the points described at the beginning of the file. Note that at least two points must be given. This will result in **`numberpoints-1`** polygonal segments.

Example:

`polyline r 1 b 2 5 0 1 2 3 0`

This will create a *closed* polygon of 4 segments, with the segments defined by vertices 0-1, 1-2, 2-3, and 3-0. Region 1 is located to the left of the segments, and the boundary condition on the right has a value of 2.

Circle boundary patch

The format for **`[extra_info]`** is:

`radius centerpoint`

where the **`radius`** is a real number and the **`centerpoint`** is an integer index to a point defined at the beginning of the file.

Example:

`circle b 1 r 2 3.40 7`

This will create a circle of radius 3.40, centered at the location defined by point 7. The sides of the circle are defined by walking counter-clockwise on the circle, so the left side is the interior, and the right side the exterior. Here, region 2 is to be meshed and is located outside the circle. The inside will not be meshed, and has a boundary condition number of 1.

Arc boundary patch

The format for **`[extra_info]`** is:

`radius startpoint endpoint`

where **`radius`** is a real number and both `startpoint` and `endpoint` are indices to points defined earlier in the file. The coordinates for the center of the arc are determined automatically. The arc is always drawn counter-clockwise, from startpoint to endpoint. Furthermore, the arc drawn will always span 180° or less. For arcs of more than 180°, use `longarc` instead.

Example:

```
arc r 3 r 2 1.0 0 1
```

This will draw a circular arc of radius 1.0, from the point with index 0 to the point with index 1. The arc is drawn counter-clockwise and spans less than 180 degrees. The arc will be an internal boundary since it has both regions 3 and 2 as neighbors.

Long arc boundary patch

The format here is the same as for arcs; the difference is in interpretation. A `longarc` is drawn counterclockwise from `startpoint` to `endpoint`, but it always spans 180° or more. For arcs of less than 180°, use `arc` instead.

Bézier boundary patch

The format for **`[extra_info]`** is:

`startpoint endpoint controlpoint1 controlpoint2`

where all the parameters are indices to points defined at the beginning of the file. The Bézier curve will be a smooth curve with a tangent in the direction of the vector defined by **`controlpoint1-startpoint`** at its start, and a tangent in the direction of the vector **`endpoint-controlpoint2`** at its end.

Example:

```
bezier r 1 b 2 4 6 8 10
```

This will create a Bézier curve defined by points with indices 4, 6, 8, and 10, and will have region 1 to its left, and the domain boundary on its right, with a boundary condition of 2.

Spline boundary patch

The format for **`[extra_info]`** is the same as for **`polyline`**, that is:

`numberpoints points`

where **`numberpoints`** is the number of points used to describe cubic interpolated spline. Then follow **`numberpoints`** indices to the points described at the beginning of the file. Note that at least three points must be given (otherwise the spline simply is a line). The spline is created with "no-moment" boundary conditions, i.e. the second derivative of the curve is zero at both ends. The curve will pass through all the points given.

Example:

```
spline b 2 r 4 10 0 1 2 3 4 5 6 7 8 9
```

This will create an open cubic spline, starting at point 0 and ending at point 9. The curve will also pass through points 1..8. Region 4 is located to the right of the spline, and the boundary condition on the left has a value of 2.

The syntax for periodic boundary patch declarations, which follow the patch descriptions in the file, is

`periodic patch1_index patch2_index`

where the indices identify two patches that will be treated as periodic; presently, only segment patches are supported in this context. Patches need not be parallel or of the same length; a general linear mapping is computed automatically to map one segment onto the other.

A sample input file, a picture of the underlying geometry, and the mesh generated from this input using `tri -r 2 -g 2` are shown in Figure 3.1. The input contains eight points and six patches, four in a single polyline, one circle, and one Bézier curve. This input file is `square-circhole-bezier.bdry` in `examples/2D`.

```
# Lines beginning with # are comments.
8 6
-0.2 0.0
0.5 0.0
1.0 0.5
1.0 1.0
-0.2 1.0
0.5 0.5
0.72 0.0
1.0 0.28
polyline r 1 b 1 5 2 3 4 0 1
circle b 2 r 1 0.45 5
bezier r 1 b 1 1 2 6 7
```

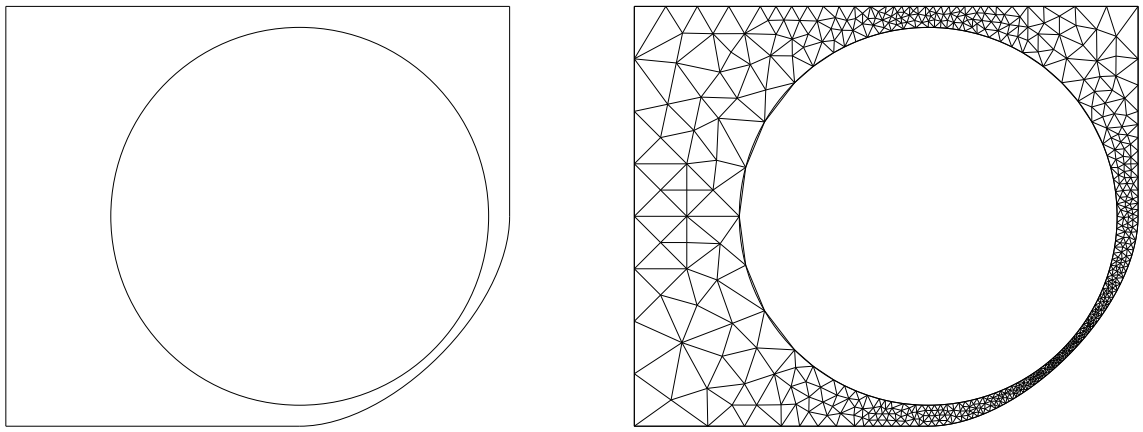

Figure 3.1: Sample 2D boundary data file and geometry.

3.2 Input and output mesh file formats

GRUMMP has the capability to input and output two-dimensional meshes in user-defined formats. To take advantage of this feature, the user must create a file called `user2d.template` in the directory `GRUMMP/src/IO_generator`. This file is processed by the programs `in2d_gen` and `out2d_gen` to produce C++ functions which, respectively, read and write two-dimensional meshes with the format described in `user2d.template`. These functions are added to the GRUMMP two-dimensional mesh manipulation library, `libGR_2D`, the next time this library is remade. A default template file, `default2d.template`, is used if `user2d.template` does not exist. Step-by-step, then, the process for setting up user-defined file formats for two-dimensional meshes is:

1. Edit `GRUMMP/src/IO_generator/user2d.template`.
2. Return to the `GRUMMP` directory.
3. Type `make`.

When data is read by the GRUMMP executables, conversion of the input connectivity data to an internal format is performed automatically. There are two valid ways to specify connectivity. First, the data file can provide both face-cell and face-vertex connectivity. Second, cell-vertex connectivity can be provided for the interior of the mesh along with boundary condition and vertex data for each boundary face. In either case, any redundant data is verified to ensure consistency.

The remainder of this section describes the format of `user2d.template`. On systems that have `flex`, `in2d_gen` and `out2d_gen` are not case sensitive; upper and lower case may be mixed indiscriminately. Systems using `lex` require lower case, because some `lex` implementations can not produce case insensitive parsers. `configure` will warn you if your system seems to be using `lex`.

The following description includes tokens in mixed case for readability. A "^" preceding a token indicates that it must begin a new line in the template file. "Whitespace" refers to any number of consecutive spaces or tabs within a line.

Token Description

^`newfile` Specifies the name of in input/output file to open. This token must be followed by whitespace and a string giving the extension to append to the *`basefilename`* (given on the command line). The file name extension is terminated by whitespace or the end of the line; if the extension is terminated by whitespace, anything following that whitespace is ignored.

Note that the first line of the template file must ***always*** be a `newfile` line.

^`Fortran` This directive indicates that numbering of mesh entities should begin with 1 instead of 0. If `Fortran` appears alone on a line of the file, vertices, cells, faces, and boundary faces are all treated this way. If `Fortran` is followed by one or more of `verts`, `cells`, `faces`, or `bfaces`, then only the entities specified are numbered in this way. A file may contain more than one `Fortran` line; the effects of multiple lines are cumulative. **Note that only data read *after* a `Fortran` directive is affected by that directive.** For this reason, it is recommended that all `Fortran` declarations be placed at the beginning of the template file (just after the initial file name).

`NVerts`

`NFaces`

`NCells`

`NBFaces`

`NIntBFaces`

These indicate that the number of vertices, faces, cells, or boundary faces in the mesh should be read or written. In each case, the token can be followed by whitespace and an integer to specify the width of the field used to read/write the number of entities. Use of field widths for input is not recommended, as this introduces the possibility of misreading data.

`^verts:` Read/write data for each vertex in the mesh. There are four data specifications for vertex data.

- `index` Read/write the number of the vertex within the mesh, starting with 0 and running to `NVerts-1`. An integer field width can be specified.
- `x`
- `y`
- `z` Coordinates of the vertex. Input/output field width `W` and number of significant figures `S` can be specified by following the token with whitespace and `W.S`.
- `coords` Read/write `x`, `y` and (in 3D) `z` (in that order). Field width and significant figures can not be specified

`^faces`: Read/write data for each face in the mesh. There are seven data specifications for face data.

- `index` Read/write the number of the face within the mesh, starting with 0 and running to `NFaces-1`. An integer field width can be specified.
- `cellA`
- `cellB` The cells incident on the face. See Figure 3.2 for orientation of cells and vertices with respect to an edge. Note that faces on the boundary will read/write the boundary condition (negating the integer BC type) in place of the missing cell. An integer field width can be specified.
- `cells` Read/write `cellA` and `cellB`, in that order. Field width can not be specified.
- `vertA`
- `vertB` The vertices at the ends of the face. Again, see Figure 3.2 for orientation of cells and vertices with respect to an edge. An integer field width can be specified.
- `verts` Read/write `vertA` and `vertB`, in that order. Field width can not be specified.

`^cells:` Read/write data for each cell in the mesh. There are nine data specifications for cell data.

- `index` Read/write the number of the cell within the mesh, starting with 0 and running to `NCells-1`. An integer field width can be specified.
- `faceA`

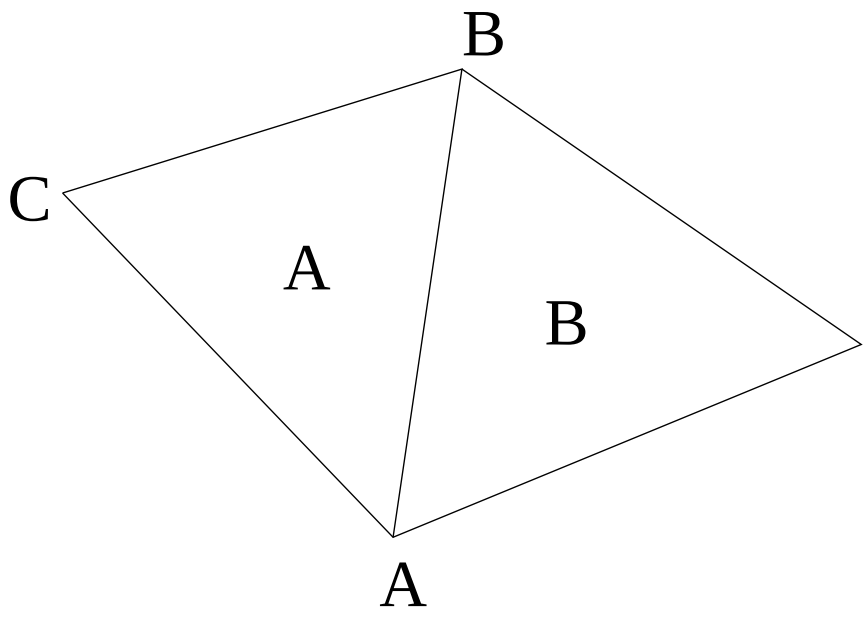

Figure 3.2: Orientation of vertices and cells as assumed by GRUMMP I/O routines.

`faceB`

`faceC` The faces bounding the cell. The order of faces is arbitrary (that is, the faces should not be assumed to be ordered cyclically). An integer field width can be specified.

`faces` Read/write `faceA, faceB` and `faceC`, in that order. Field width can not be specified.

`vertA`

`vertB`

`vertC` The vertices of the cell. These are labeled cyclically. Cell A in Figure 3.2 gives an example. Field width can be specified.

`verts` Read/write `vertA, vertB,` and `vertC`, in that order. Field width can not be specified.

`region` Read/write the region tag for a cell.

`^bdryfaces:`

Read/write data for each boundary face in the mesh. There are six boundary specifications for boundary face data.

`index` Read/write the number of the boundary face within the mesh, starting with 0 and running to `NBFaces-1`. An integer field width can be specified.

`BC` Integer indicating boundary condition. Must be greater than 0. Field width can be specified.

`face` Face index for the face corresponding to the boundary face. Field width can be specified.

`vertA`

`vertB` Vertex indices for the boundary face. Orientation is shown in Figure 3.3. Field width can be specified.

`verts` Read/write `vertA` and `vertB`, in that order. Field width can not be specified.

`^intbdryfaces:`

Read/write data for each boundary face in the mesh. There are six boundary specifications for boundary face data.

`index` Read/write the number of the boundary face within the mesh, starting with 0 and running to `NBFaces-1`. An integer field width can be specified.

`BC` Integer indicating boundary condition. Must be greater than 0. Field width can be specified.

`faceA`

`faceB`

`faces` Face indices for the faces corresponding to the internal boundary face; the same restrictions and privileges apply as for cells.

`vertA`

`vertB` Vertex indices for the boundary face. Orientation is as shown in Figure 3.2. Field width can be specified.

`verts` Read/write `vertA` and `vertB`, in that order. Field width can not be specified.

`^#` At the beginning of the line, denotes a comment. Ignored when generating input routines. Comments in the template are echoed to the output file unless they begin with ##, in which case they are ignored.

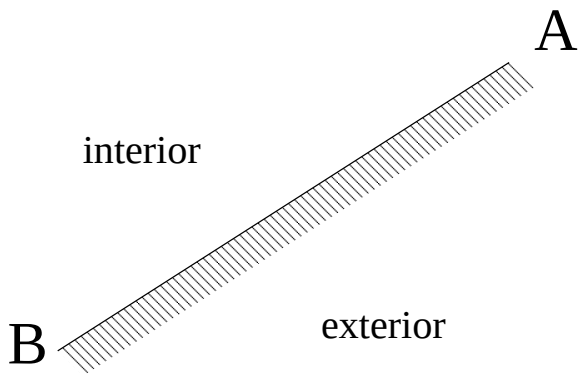

Figure 3.3: Orientation of boundary data in two dimensions.

text in quotes Echoed literally into the output file, with one exception: a `\"` in the template file becomes a `"` in the output file. In the input file, the text that is put into the output file is *required* to appear. For example, the template line

```
verts:  "x" x "y" y "\"x and y\""
```

produces the following output for a vertex at (0.5, 0.7):

```
x 0.5 y 0.7 "x and y"
```

Also, any input line compatible with this template will have the same non-numeric text as the output example just given.

other text This text is also echoed literally to the output file. For example, TecPlot requires descriptive information for each mesh zone, which for triangular meshes can be provided by including

```
ZONE N=nverts, E=ncells, F=FEPOINT, ET=TRIANGLE
```

at the appropriate place in the template file (see `IO_generator/TECPLOT2D.template`).
Caveats:

- Text containing quotes must be treated as shown above.
- A # at the beginning of a line must be quoted or it will be treated as a comment.
- Numbers following keywords should be quoted if they are not meant to specify a field width.

An example template file (in fact, the default two-dimensional template, `default2d.template`) is shown in Figure 3.4.

```
newfile mesh
ncells nfaces nbfaces nverts
verts: coords
faces: cells verts
bdryfaces: face bc verts
cells: region
```

Figure 3.4: Example template file for 2D I/O.

Three-dimensional I/O templates are quite similar to two-dimensional templates, conceptually. Obviously, in this case there are three spatial coordinates, x, y and z. Also, faces, boundary faces, and internal boundary faces can now specify a `vertC`; cells can specify a `vertD`, a `faceC` and a `faceD`.

The orientation of cells and vertices around a face is such that, if the right hand rule is used to trace vertices A, B, and C, then one's thumb points towards cell B, as shown in Figure 3.5. For a boundary face, the orientation of the vertices is such that the interior of the domain is where cell B is in Figure 3.5. The default three-dimensional template (`default3d.template`) differs from its two-dimensional counterpart only in that the file suffix given is `vmesh` instead of `mesh`.

In both two and three dimensions, an alternative template file (`mr[23]d.template`) is also provided for output of meshes with internal boundaries; note that `coarsen[23]d` and `meshopt[23]d` can not currently read multi-domain files.

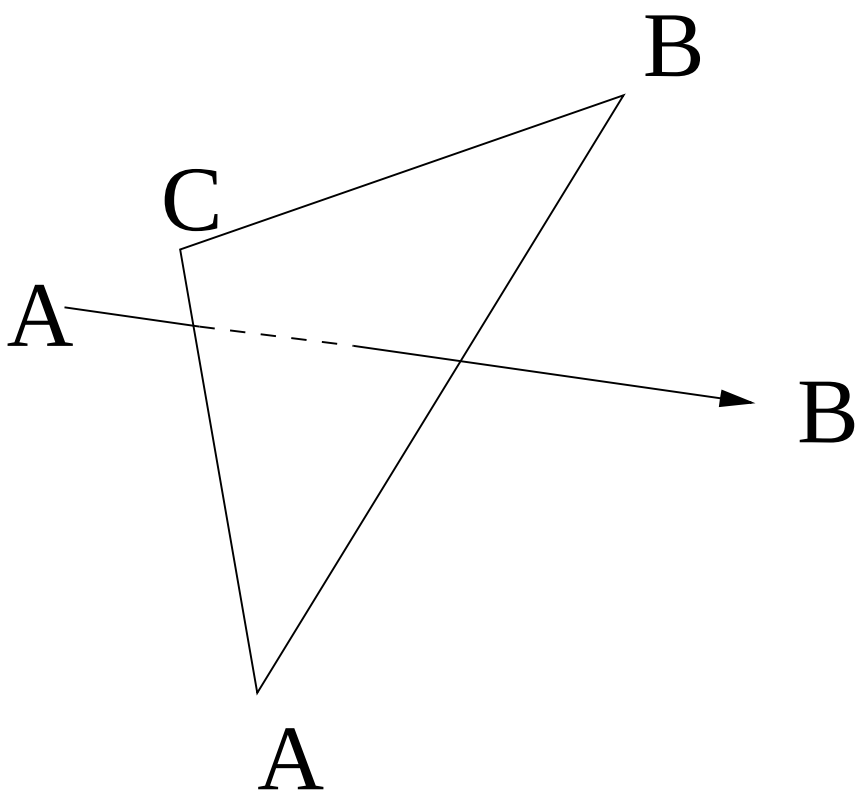

Figure 3.5: Orientation of cells and vertices for a three-dimensional face.

3.3 Three-dimensional geometry (`.bdry`) files

The preferred way to specify a three-dimensional domain for meshing is using a geometry (`.bdry`) file. This file format supports simultaneous and consistent meshing of multiple sub-domains and is extensible to non-polyhedral boundaries. This file format is conceptually the same as the two-dimensional geometry file format, so most of the description of the format is the same.

The first line of the file contains the number of vertices and number of boundary sections in the file. The next `NVerts` lines give vertex coordinates. The remainder of the file describes the boundary geometry. A sample input file and a picture of the underlying geometry are shown in Figure 3.6.

Each boundary data entry begins with a tag describing the kind of data it contains. At present, the only valid tag is **`polygon`**.

Next, the relationship between the boundary entity (BE) and the domain is established. For entities lying on the bona fide domain boundary, a boundary condition must be specified, along with an integer tag ($1 \leq$ tag ≤ 126) identifying which interior region is adjacent to the BE. Entities lying within the domain necessarily separate two regions with different ID tags; both tags must be specified.

Finally, geometric data required to describe the BE must be given. This data is not restricted to a single line

in the input file.

```
# This geometry is a cube diced into four regions.
# Data from file: examples/3D/simple/hex-multi.bdry
#
# 12 vertices, 17 bounding polygons
#
12 17
#
# Vertex coordinates
#
-1 -1 -1
 1 -1 -1
 0  0 -1
 1  0 -1
-1  1 -1
 1  1 -1
-1 -1  1
 1 -1  1
 0  0  1
 1  0  1
-1  1  1
 1  1  1
#
# Now the geometry description
#
polygon b 1 r 1 3 0 2 4
polygon b 1 r 1 3 6 10 8
polygon b 1 r 1 4 0 4 10 6
polygon b 1 r 2 3 2 5 4
polygon b 1 r 2 3 8 10 11
polygon b 1 r 2 4 4 5 11 10
polygon b 1 r 3 4 0 3 5 2
polygon b 1 r 3 4 6 8 11 9
polygon b 1 r 3 4 5 3 9 11
polygon b 1 r 4 3 0 1 3
polygon b 1 r 4 3 6 9 7
polygon b 1 r 4 4 1 0 6 7
polygon b 1 r 4 4 3 1 7 9
polygon r 1 r 2 4 2 4 10 8
polygon r 2 r 3 4 2 5 11 8
polygon r 1 r 3 4 0 2 8 6
polygon r 3 r 4 4 0 3 9 6
```

Figure 3.6: Sample 3D boundary data file.

`polygon` geometric data

A **`polygon`** is specified by giving the number of vertices in the polygon, followed by the indices of the vertices, in order. These indices begin from 0.

3.4 Three-dimensional surface mesh (`.smesh`) files

Surface mesh files may also be specified in the form of a surface triangulation. This is particularly convenient when the surface mesh is created by some other piece of software.[1] File format for surface triangulation input can be summarized using a pseudo-template file (see Section 3.2), where `nsurfs` is the number of disconnected closed surfaces.

```
face-based ncells nfaces nsurfs nverts
verts: coords
faces: cells verts bc
```

Note that this format is hard-coded, and not a true template. Also, multi-domain geometries can not be input using `.smesh` files, because the file format is not expressive enough to allow non-manifold surface meshes. Geometry input will be rewritten at some point in the future to allow easier extensibility to new geometry types; when this happens, support for the `.smesh` file format may vanish, so new input files should use the `.bdry` format.

3.5 Three-dimensional ASCII stereolithography (STL) files

Currently, GRUMMP only supports the ASCII STL format. These files consist of a one line header and footer bookending a collection of facet blocks:

solid *name*
  facet normal n_x n_y n_z
    outer loop
      vertex $v1_x$ $v1_y$ $v1_z$
      vertex $v2_x$ $v2_y$ $v2_z$
      vertex $v3_x$ $v3_y$ $v3_z$
    endloop
  endfacet
endsolid *name*

Words in boldface are keywords and must be lower case. Spacing is important: spaces between keywords must be as shown. Normal and coordinate components are interpreted as floating-point numbers.

[1] Future plans include automatic or semi-automatic splining of curved surfaces input as triangulations. If you're interested in hurrying this along, send a post-doc. :-)

3.6 Three-dimensional volume mesh input and output (`.vmesh`) files

Three-dimensional mesh files have the format that one would find from an automatically-generated output driver using the following template file:

```
newfile vmesh
ncells nfaces nbfaces nverts
verts: coords
faces: cells verts
bfaces: face bc verts
cells: region
```

3.7 Mesh quality (`.qual`) files

Mesh quality is assessed at several points during mesh generation and improvement. Each time the quality is evaluated, each cell is queried to determine its quality.[2] The results are sorted into bins and a count kept for each bin. In addition, the overall maximum and minimum quality values are computed. After all quality values have been computed and binned, the count for each bin is divided by the total number of quality values; therefore, each bin has a value that represents the fraction of all quality values that fall within that bin.

When mesh quality statistics are written to a file, the quality measure used and the maximum and minimum quality for each evaluation are included at the top of the file in a format which is commonly ignored as a comment by plotting software. Each line after that contains the quality value at the center of a bin and the fraction of quality values that fell within that bin for each evaluation. An example of a mesh quality file is shown in Figure 3.7.

3.8 Status message (`.msg`) files

As the GRUMMP library routines are executed, they generate a number of status and informational messages that are written to standard error (generally the screen). In addition, a more extensive set of messages are written to the file *`basefilename`*`.msg`. While these messages are unlikely to be of tremendous interest to users of the GRUMMP executables, they are often useful in debugging and therefore may be of interest to developers using the GRUMMP libraries.[3]

3.9 Scattered data interpolation files

For scattered data interpolation, there are two input files and one output file.

[2] Measures which compute all angles for a cell obviously have more than one value for a cell.

[3] Be aware that when GRUMMP is configured using `--with-debug`, the `.msg` file may be quite large; I've filled the disk on my laptop a time or two, although I've since reduced the number of messages routinely written.

```
# Quality measure:  all dihedral angles
#
# Qual eval 1.  Min qual =     0.657175  Max qual =      178.882
# Qual eval 2.  Min qual =      3.36452  Max qual =      172.377
# Qual eval 3.  Min qual =       5.1961  Max qual =       164.55
# Qual eval 4.  Min qual =       5.1961  Max qual =      164.254
#
  3  0.00109629 0.000651634 1.51543e-05 1.51543e-05
  9  0.00429507  0.00269746 0.000242468 1.51543e-05
 15  0.00731363  0.00521307  0.00193975 0.000878948
 21   0.0120442    0.011396   0.0067588  0.00597078
 27   0.0187421   0.0198369   0.0143966   0.0137449
 33   0.0305761   0.0354913   0.0319604   0.0315663
 39   0.0504295   0.0569952   0.0546311   0.0531612
 45   0.0718598   0.0760593   0.0758471   0.0759532
 51   0.0874332   0.0891071   0.0918046   0.0923047
 57   0.0919986   0.0894102   0.0950476   0.0995636
 63   0.0896558   0.0845154   0.0965024   0.0962448
 69   0.0883192   0.0836061    0.095093   0.0949112
 75   0.0809005   0.0757562   0.0837425   0.0866218
 81   0.0740524   0.0699673   0.0745439   0.0747409
 87    0.067805    0.065815   0.0664666   0.0665272
 93   0.0643059   0.0644056   0.0581015   0.0570104
 99   0.0458491   0.0466903   0.0446142   0.0454628
105   0.0360726   0.0368097   0.0353549   0.0339001
111   0.0269118   0.0292174    0.025247   0.0244438
117    0.018667   0.0217767   0.0190338   0.0190338
123   0.0108278   0.0129114    0.012472   0.0124568
129  0.00815462  0.00971389  0.00741044  0.00769837
135  0.00548147  0.00577378  0.00453113  0.00401588
141  0.00301856  0.00325817   0.0023186  0.00213675
147  0.00183216  0.00148512  0.00101534 0.000954719
153 0.000901063 0.000742559 0.000727405 0.000545554
159 0.000720851 0.000348548 0.000136388  0.00010608
165 0.000390461 0.000227314 4.54628e-05 1.51543e-05
171 0.000270319 0.000121234           0           0
177 7.50886e-05           0           0           0
```

Figure 3.7: Example quality output file

The point input file also contains the data for each vertex. This file has the following format (written as a pseudo-template file).

```
nverts ndata_per_vert
verts: coords data1 data2 ...
```

The input file of target points is even simpler:

```
nverts
verts: coords
```

The output file is also quite simple:

```
verts: coords new_data1 new_data2 ...
```

Part II

Algorithms Used in the GRUMMP Libraries

Chapter 4

Mesh Generation

GRUMMP generates meshes by using Delaunay refinement. As a class, Delaunay refinement methods generate Delaunay meshes (meshes in which the ball defined by the vertices of a given cell contains no other vertices in its interior) of high quality by careful choice of new locations to insert vertices. The schemes in GRUMMP follow the work of Ruppert [14, 15] and Shewchuk [16], with several significant improvement of our own in the areas of cell size and grading control [11] and meshing from curved boundaries [1].

4.1 Two dimensional meshing

In two dimensions, GRUMMP follows Shewchuk's modification to Ruppert's scheme fairly closely, except that we exercise more precise control over cell size and grading and extend the scheme to produce guaranteed-quality meshes from domains with curved boundaries.

Ruppert's scheme [14, 15] begins with a constrained Delaunay triangulation.[1] The mesh quality is improved through point insertion. Points are inserted at the circumcenter of badly-shaped cells, cells that have an angle less than $\theta_{\min}$, unless they *encroach* on a boundary edge. A vertex *encroaches* on an edge when that vertex is located inside the circle with the edge as its diameter; this circle is called the *diametral circle*. If a proposed new point encroaches on any boundary edge, that vertex is not inserted. Instead, the encroached boundary edge(s) is(are) bisected. This process is repeated until all cells are well-shaped. Ruppert was able to show that this algorithm always terminates, and results in a mesh with minimum angle $\theta_{\min} \approx 20.7^\circ$.

Shewchuk [16] showed that a $\theta_{\min} = 25.7^\circ$ is possible if *diametral lenses* rather than diametral circles are used to determine if there is encroachment. The difference between the diametral circle and diametral lens is shown in Figure 4.1. In this variant of the algorithm, interior vertices lying inside the diametral circle of a boundary edge are deleted when that edge is split. The bound on $\theta_{\min}$ is not tight; in practice, $\theta_{\min}$ can be set to 30° and the algorithm will still terminate.

[1] A constrained Delaunay triangulation is a triangulation in which the Delaunay criterion is only applied to vertices that are visible to a triangle. A vertex is visible to a triangle if there are no boundary patches between them.

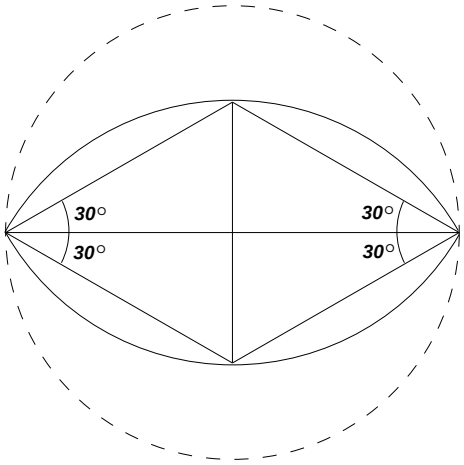

Figure 4.1: Comparison between a diametral circle (dashed) and diametral lenses. Diametral lenses allow points to be inserted closer to boundary edges.

4.1.1 Initial discretization

Ruppert's algorithm can be started either with a Delaunay triangulation or a constrained Delaunay triangulation. The latter does not pose a problem because Ruppert's original encroachment rule guarantees that no vertex will be inserted outside a boundary edge.[2] A Delaunay triangulation containing all the boundary points inside a larger bounding box is first created. Boundary edges are recovered by swapping; this approach always succeeds for two-dimensional polygonal boundaries. The triangles lying outside the domain are then removed, leaving a constrained Delaunay triangulation. Initial triangulations are created by a `Mesh2D` constructor in `src/2D/Mesh2D.cxx`, using geometric information stored in a geometry object of type `Bdry2D`.

4.1.2 Point insertion

GRUMMP uses the Delaunay insertion method of Watson [17]. First, the code lists all cells that contain the new vertex in their circumcircle. These cells are then removed from the mesh, and the faces of the resulting hull are connected with the newly inserted point. This insertion method preserves the Delaunay nature of the mesh; no swapping is needed after the insertion. If a boundary edge is part of the hull, a check is made to ensure that the new vertex will not encroach on it. If it does, the point is not inserted. Vertices lying inside the diametral circle of the edge are removed, and the boundary edge is split at its geometric midpoint. GRUMMP uses Watson insertion for this split as well. The actual insertion code is found mostly in `src/2D/InsertWatson2D.cxx`, while the driver for Ruppert's scheme is in `src/2D/Ruppert.cxx`.

[2] The use of diametral lenses allows boundary triangles with a circumcenter outside the boundary edge to be present in the mesh. However, no vertex will ever be inserted at this location since it encroaches on the boundary edge.

4.1.3 Length scale modifications

GRUMMP uses a more flexible definition of geometric length scale than Ruppert's; for implementation details and proofs of mesh quality and size-optimality, see Ollivier-Gooch and Boivin [11]. GRUMMP computes a geometric length scale based on the *local feature size*. The local feature size was used by Ruppert to prove termination of the original algorithm, and is defined as the radius of the smallest circle centered at a point that touches two disjoint parts of the domain boundary. We defined the length scale LS in terms of the local feature size lfs as:

$$LS(p) = \min\left(\frac{lfs(p)}{R}, \min_{\text{neighbors } q_i} LS(q_i) + \frac{1}{G}|\vec{q}_i - \vec{p}|\right) \tag{4.1}$$

where both R and G are constants ≥ 1, and points q_i are neighbors to point p. The first constant, R, controls the ratio of input feature size to final mesh boundary edge length, with finer boundary discretization for larger values of R. The other constant, G, is used to control how rapidly the cell size can change with distance. This is an explicit imitation and generalization of the grading properties of the local feature size. A larger value of G results in slower increase in cell size over the same distance. The value of LS is stored at every vertex location.

Ruppert's scheme can be easily modified to split cells that are too large according to the definition of length scale in Equation 4.1, in addition to splitting those that are badly shaped. A cell is considered too large whenever the ratio of its circumradius to the average LS of its vertices is greater than $\frac{\sqrt{2}}{2}$.

Length scale is computed and manipulated by code in `src/base/Length.cxx`.

4.1.4 Curved boundaries

This section is a digest of the description found in Boivin and Ollivier-Gooch [11]. The main result of that paper was to demonstrate that it is possible to generate a guaranteed-quality triangular mesh in a domain with curved boundaries. Modifications to the meshing code itself are relatively slight, although geometric modeling is significantly more complex.

Interested readers are encouraged to refer to that paper for full details, including implementation issues regarding the curve types implemented in GRUMMP.

Generic geometry framework

To enable meshing from general curved boundaries, GRUMMP uses a framework in which the mesh generation code makes no assumptions about the underlying geometry of boundary patches. This implies a generic interface between mesher and geometry, in which the mesher only needs the results of several geometric queries. This is illustrated in Figure 4.2.

Whenever GRUMMP needs information about the boundary, a "question" is passed on to the proper type of boundary patch. Each boundary patch type knows how to answer all of these questions, and the answer is then passed back to the algorithm. This provides a transparent access to potentially any type of boundary patch. Using object-oriented programming, this generic interface can be implemented by using a common

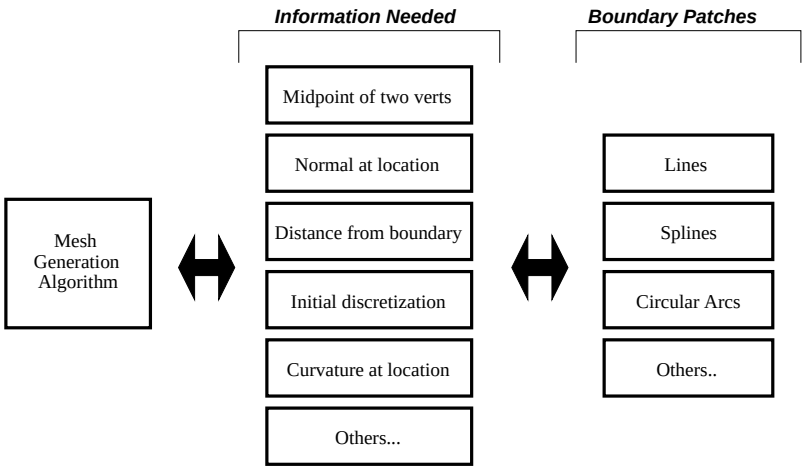

Figure 4.2: Framework used for the implementation of generic boundaries

base class for all boundary data, with implementation of specific geometric queries in derived boundary data classes.

The information required for the successful implementation of Ruppert's algorithm — curve midpoint, curvature, and original discretization information — is described in Sections 3.2 to 3.6. Several other queries are required to determine the appropriate mesh length scale *LS*, but will not be discussed further here.

So far, classes for lines, circles, arcs, cubic parametric curves, and interpolated splines have been written. New types of boundary patches can be added by providing the proper "answers" for the given boundary patch. These patch definitions can be found in `src/2D/Bdry*.cxx`.

Measuring how curved the boundary is

GRUMMP must be able to work with curved as well as linear patches, so a new way of determining where splits happen along a boundary patch is necessary. We first make the observation that patches with little orientation change need few, long edges for accurate geometric representation. Linear patches have no orientation change; they can be represented accurately with just one edge. In contrast, regions of a curve with a large change in orientation require a greater number of shorter edges. We must also make sure that small amplitude sine-like curves are discretized appropriately. This suggests we should use the total variation of the tangent angle of a curve to determine where to split a boundary patch.

The total variation $TV(\theta)$ is defined in the following way:

$$TV(\theta) = \int |\, d\theta \,| \tag{4.2}$$

Note that there is no need to compute the integral; one simply needs to compare the orientation of the curve's tangent vector at carefully chosen points along the boundary patches to get the exact value of $TV(\theta)$. More details are given for each type of boundary patches in Boivin and Ollivier-Gooch [1].

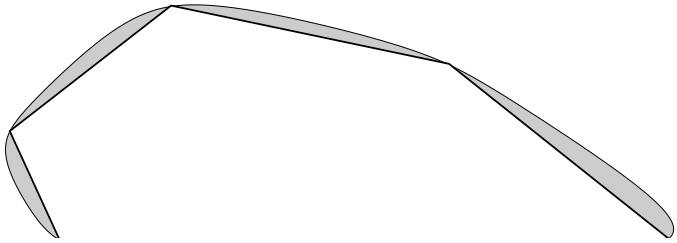

Figure 4.3: Arbitrary original discretization of a spline. No vertex should be inserted in the shaded areas.

Initial discretization with curved boundaries

To obtain the initial Delaunay triangulation, each boundary patch must be initially discretized in some way. Since the exact shape of the boundary is only known by the boundary patches, the initial discretization of the corresponding curve must be computed by the patches themselves. At this point in the meshing process, we represent curves with as few edges as possible in order not to introduce artificial small features in the mesh. However, we must make sure that a valid and exact representation of the domain will be obtained and that the rules regarding the location of points inside the diametral lenses are also followed.

An arbitrary discretization of a spline curve is shown in Figure 4.3. We wish to triangulate outside (above) the curve. Ruppert's scheme guarantees that no vertex will be inserted inside (below) the boundary edges. We also want to make sure that no vertex will be inserted in the regions inside the curve but outside of the boundary edges (the shaded area in Figure 4.3). This is to prevent an invalid discretization, as the vertex inserted in the shaded area would ultimately lie outside the domain once the boundary is well-resolved.

This area can be protected by making sure that the diametral lenses of the boundary edges completely include the curve boundary. Since points are never inserted inside the diametral lenses, this will protect the shaded region from point insertion. It is easy to show that the maximum allowable total variation in orientation between vertices is 30°. GRUMMP boundary patches determine their initial discretization by arranging vertices at equal intervals of $TV(\theta)$ so that the maximum orientation change condition is met.

Edge recovery

Due to the very coarse representation of the boundary patches during edge recovery, some precautions must be taken in order to get a valid initial constrained Delaunay triangulation. The edge recovery process must be modified since simple recovery through swapping will fail in some cases. Judicious use of vertex insertion is required in these cases to obtain a valid initial triangulation. The resulting edge recovery algorithm is summarized in Figure 4.4.

Point insertion

Point insertion in the mesh, as well as on the boundary, is still done using Watson's method. However, curved boundaries modify the way that boundary edges are split. Instead of splitting at the average location of the edge's vertices, the location of the new boundary vertex is determined by the boundary patch itself.

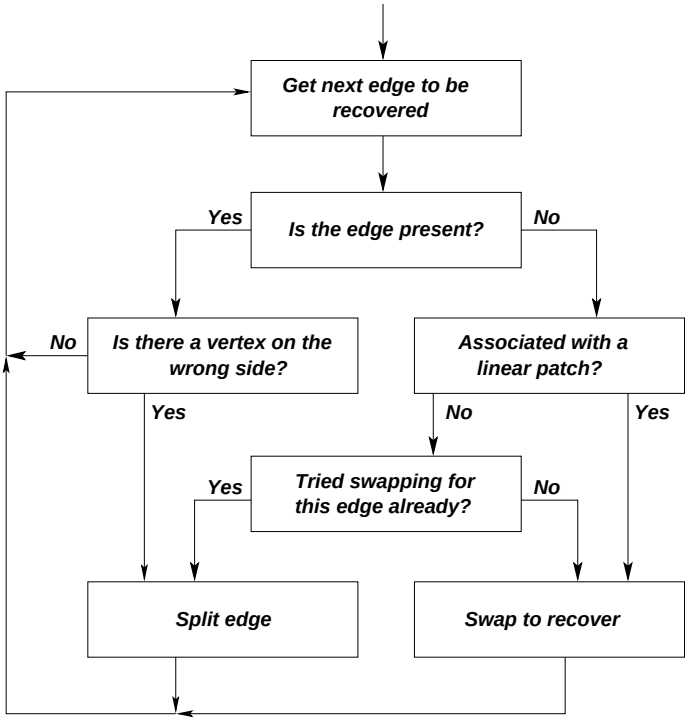

Figure 4.4: Procedure to follow to recover boundary edges

The "midpoint" between two vertices is now found using the total variation of the tangent angle. The general technique is to first find the total variation of the tangent angle between the boundary edge's vertices a and b. The midpoint c will be located at the point on the curve where $TV(\theta)$ between a and c and between c and b is equal. This ensures that the new point is always located on the boundary and that regions of the curve with higher curvature will be discretized with more edges.

If the curvature over a given boundary edge is (almost) zero, the orientation change is negligible. In these cases, the split is made according to arclength. This ensures linear patches are split in the same fashion as before, and it also handles curves that have particularly flat regions.

Care must be taken when two curved patches are near each other, because the discretization of one patch may intersect the neighboring patch. Insertion on the latter can result in points outside the domain unless the patches are both split, and in the proper order.

Length scale modifications

Whenever a boundary edge is split, the length scale $LS(p)$ for the new boundary vertex needs to be computed using Equation 4.1. For this, we need the local feature size $lfs(p)$ at the new point p to take into account the curvature of the boundary. We define the local feature size for curved boundaries, lfs_c to be:

$$lfs_c(p) = \min\left(\rho\left(p\right), lfs\left(p\right)\right) \tag{4.3}$$

where $\rho(p) = \frac{1}{|\kappa(p)|}$ is the radius of curvature at point p. The radius of curvature therefore provides a ceiling on the value of the local feature size on the boundary. By using the radius of curvature, there will be an equal number of points per radian on the curve as per gap between objects.

4.2 Three-dimensional meshing

In three dimensions, GRUMMP follows Shewchuk's scheme [16], except that again we exercise more precise control over cell size and grading.

4.2.1 Initial tetrahedralization

In concept, GRUMMP creates an initial tetrahedralization in much the same way as an initial triangulation is created in 2D: all vertices of the surface discretization are inserted into a mesh inside a large box, then the surface is recovered and tetrahedra outside the domain are removed. The difficulty lies in the second step, "the surface is recovered". In two dimensions, swapping alone is adequate. In three dimensions, this is not always true. In his thesis, Shewchuk shows that a surface mesh for which spheres protecting boundary segments and triangles are point-free must have a constrained Delaunay tetrahedralization; that is, that the surface can be recovered. In practice, this can require insertion of a significant number of points, although heuristics can improve significantly on the sometimes dismal requirements of theory. This is currently still a weak spot in GRUMMP: surface recovery is neither as efficient nor as robust as it should be, with particular challenges for surface meshes with small angles.

4.2.2 Point insertion

Following Ruppert, Shewchuk defines a *boundary segment* as an edge present in the input geometry. If a boundary segment is divided into parts by subsequent point insertion, each part is called a *boundary subsegment*. Shewchuk also defines a *boundary facet* as a planar polygonal surface in the input, bounded by boundary segments. When a boundary facet is divided, either by triangulation or addition of points in its interior, the parts are referred to as *boundary subfacets*. A triangular boundary subfacet is encroached if a point lies within the *equatorial sphere* of the triangle: the unique sphere having the circumcircle of the triangle at its equator. A boundary subsegment is encroached if a point lies within the unique sphere that has the subsegment as its diameter.

Shewchuk's scheme can be summarized (minus some details) as follows (including GRUMMP's addition of size and grading control, as described in two dimensions):

1. If a tetrahedron is badly shaped — if its shortest edge is too small in comparison to its circumradius — or too large, then insert a point at its circumcenter UNLESS that point would encroach on any boundary subfacet or subsegment, in which case the cell circumcenter is not inserted. Instead, encroached boundary entities are split; if the original cell still exists after splitting boundary entities, then the cell is split.

2. If a vertex encroaches on a boundary subfacet **and** the normal projection of the vertex into the plane of the subfacet lies inside the subfacet, split the subfacet. Subfacets are split by inserting a point at their circumcenter UNLESS that point would encroach on any subsegment, in which case, the subfacet circumcenter is not inserted. Instead, all encroached subsegments are split. If the subfacet survives subsegment splitting, the subfacet is split afterwards. Before a subfacet is split, all points inside its equatorial sphere (not the equatorial lens, which is point-free) are deleted from the mesh.

3. If a boundary subsegment is encroached, it must be split. If the newly-inserted vertex encroaches on any other subsegments, those subsegments should be split as well. Care must be taken with handling of small input angles to prevent infinitely recursive insertion.

Subsegment splitting takes priority over subfacet splitting, which takes priority over bad cell removal. This variant of Shewchuk's algorithm can be shown to limit the ratio of circumradius to shortest edge length for good grading to 2. Although Shewchuk's scheme has an agreeably strong bound on mesh quality, it still allows some sliver tetrahedra, which are often problematic for solution of partial differential equations. Fortunately, post-processing the mesh by swapping and smoothing eliminates nearly all such tets from the mesh.

GRUMMP implements Shewchuk's scheme by using a priority queue. Tetrahedra are given a priority based on size and shape. For each tetrahedron, a size measure $M_L = \frac{2r}{\sqrt{3LS}}$ and a shape measure $M_S = \frac{\sqrt{6}l_{\min}}{4r}$ are computed. The size measure is the ratio of the circumdiameter of the tetrahedron to the average of the length scale at its vertices, with a constant factor so that a cube can be tetrahedralized with no interior points. The shape measure expresses the ratio of shortest edge length to circumradius, normalized so that all values lie between 0 and 1, and an equilateral tetrahedron has quality 1. If the tetrahedron is too large $(M_L > 1)$, then the tetrahedron is assigned a priority of $-M_L + M_S$. Otherwise, the priority is M_S. In practice, tetrahedra much smaller than the length scale are not queued for splitting by GRUMMP, regardless of quality, to prevent infinite recursion near small dihedral angles in the surface mesh, where theory provides no guarantees of termination (and practice often fails also).

Encroached boundary facets and boundary segments are also included in the queue, with priorities set to large negative values so that traversing the queue from lowest numerical priority value to highest follows the insertion rules described above.

Watson insertion is used in 3D as well as in 2D, and encroached boundary entities can be identified easily before the mesh is changed. New encroached entities are added the insertion queue (with a priority slightly less urgent than other similar encroached entities, to prevent infinite recursion). Before insertion actually occurs, a check is done to ensure that the entity at the head of the queue (be it a tetrahedron, a boundary triangle, or a boundary segment) is the one for which insertion was originally requested. If it is, insertion proceeds. If not, GRUMMP attempts to split the entity now at the head of the queue.

The queue is built and manipulated (including calls for insertion and for adding new entities to the queue) by code found in `src/base/InsertionQueue.cxx` and `src/base/WatsonInfo.cxx`. Three-dimensional Watson insertion code is found in `src/3D/InsertWatson3D.cxx`.

Chapter 5

Mesh Improvement

GRUMMP uses two main strategies for improving existing meshes (including those generated by GRUMMP): local reconnection (swapping) and smoothing. For a more complete description, including the results of numerous computational experiments in mesh improvement, see Freitag and Ollivier-Gooch [3].

5.1 Swapping

In both two and three dimensions, GRUMMP uses the standard well-established face swapping techniques. In addition, GRUMMP also implements edge swapping in 3D [3].

In two dimensions, face swapping chooses the best diagonal for the quadrilateral formed by two neighboring triangles.[1] GRUMMP can choose the diagonal to satisfy the Delaunay criterion; to minimize the maximum angle; or to maximize the minimum sine of angles (in two dimensions, this is formally equivalent to the Delaunay criterion).

In three dimensions, reconfiguration is more complex. The canonical case is exchanging two tetrahedra that share a face with three tetrahedra, as shown in the top left part of Figure 5.1. The converse swap, from three to two tetrahedra, is also possible. In addition, GRUMMP allows reconfiguration of two tets to two (T22 case in the figure); in this case, the two shaded faces must be co-planar, and swapping decisions reduce to choosing the best diagonal for the coplanar quadrilateral. If two pairs of tetrahedra in the interior of the mesh share a pair of coplanar faces, this swap is also permitted; in this case, two T22 configurations are back-to-back in the mesh. In addition to these swappable configurations, there are a number of unswappable cases, some of which are illustrated in the bottom of Figure 5.1.

Edge swapping is a more complicated procedure, replacing N tetrahedra incident a single edge by a new set of $2N-4$ tetrahedra. In the example of Figure 5.2, the edge TB is perpendicular to the page. The five tetrahedra originally incident on it ($01TB$, $12TB$, $23TB$, $34TB$, and $40TB$ are replaced by six new tetrahedra, two for each of the triangles of the (non-planar) triangulation of polygon 01234: $012T$, $024T$, $234T$, $021B$, $042B$, and $324B$.

[1] The quadrilateral must of course be convex for face swapping to be performed.

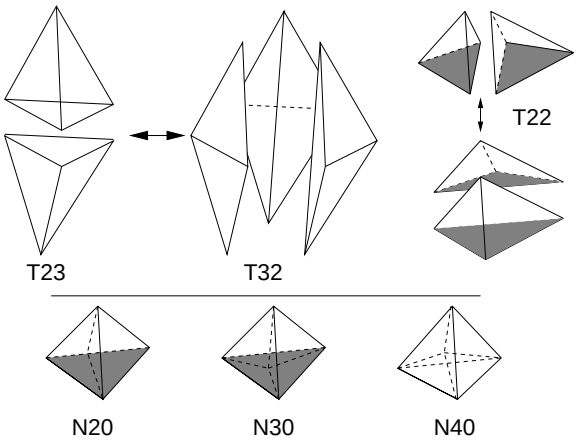

Figure 5.1: Face swapping in three dimensions

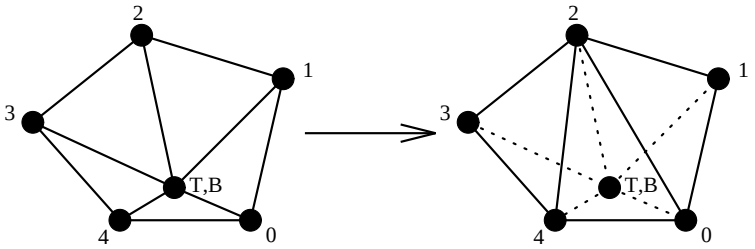

Figure 5.2: Edge swapping example

The challenge with edge swapping is that the number of possible configurations grows rapidly with the number of tetrahedra incident on the edge to be removed, as seen in the table below. In practice, the number of successful 7-for-10 swaps is very small, so GRUMMP does not explore possible swaps for more complex initial configurations.

Tets before	3	4	5	6	7
Tets after	2	4	6	8	10
Configurations	1	2	5	14	42
Tets$\times$configs	2	8	30	112	420
Unique tets	2	8	20	40	70

Clearly, checking the quality of each tetrahedron in each possible configuration is a costly undertaking; instead, GRUMMP computes the quality for each unique tetrahedron only once, then determines the quality of a given configuration by finding the minimum quality among its tetrahedra.

To simplify bookkeeping, GRUMMP takes advantage of the symmetries of the post-edge-swapping configurations and stores only a small set of canonical configurations, as shown in Figure 5.3. For each configuration, GRUMMP also stores connectivity information for the post-swap configuration.

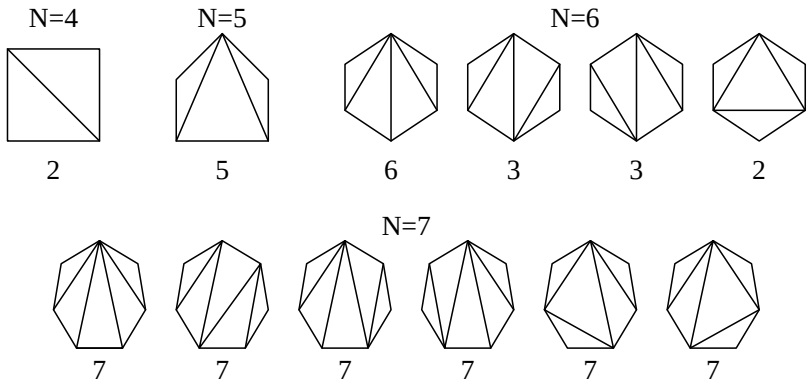

Figure 5.3: Canonical configurations for edge swapping, including repeat count.

For both face and edge swapping, swapping can be done with the goal of maximizing the minimum sine of dihedral angles or of minimizing the maximum dihedral angle. Also, face swapping can use the Delaunay criterion. The maxmin sine criterion eliminates both small and large dihedral angles, and is recommended choice in GRUMMP.

Code for swapping can be found in `src/2D/MeshOpt2D.cxx`, `src/2D/Reconnect2D.cxx`, `src/3D/MeshOpt3D.cxx`, `src/3D/Reconnect3D.cxx`, and `src/3D/InitCanon.cxx`.

5.2 Smoothing

GRUMMP uses the OptMS smoothing package from Argonne National Laboratory, which is based in large part on optimization-based smoothing techniques developed by Freitag, Jones, and Plassmann [2] and summarized in [3]. The central idea is to smooth a vertex location by seeking to optimize some local cell shape measure, such as minimum angle. This implies that each cell may contribute more than one quality value (for example, a tetrahedron has six dihedral angles). The objective function to be optimized is therefore a composite, piecewise-smooth function, as shown in Figure 5.4, with corners where the extremal quality values change. However, for most common cell quality measures, the composite objective function is convex. OptMS uses an analog of the steepest descent method, employing an *active set* of quality values — those values that are currently the worst. Step size in the steepest descent iteration is determined by predicting where the next change in active quality values will take place. This approach can be shown to be equivalent to a generalized linear programming problem and therefore the optimum can be found in linear time.

As an important practical matter, optimization-based smoothing is significantly more expensive per vertex than Laplacian smoothing. Accordingly, GRUMMP takes advantage of the floating threshold capability of OptMS. In this procedure, a Laplacian smoothing step is performed for each vertex; the updated position is used only if local mesh quality improves. If local mesh quality is not good enough, optimization is invoked. “Good enough”, in this context, means that cells incident on the vertex have a worst angle no more than 5° better than the worst angle in the mesh from the previous smoothing pass. This floating threshold procedure results in mesh improvement statistics comparable to full optimization, but at a cost nearly as low as Laplacian smoothing.

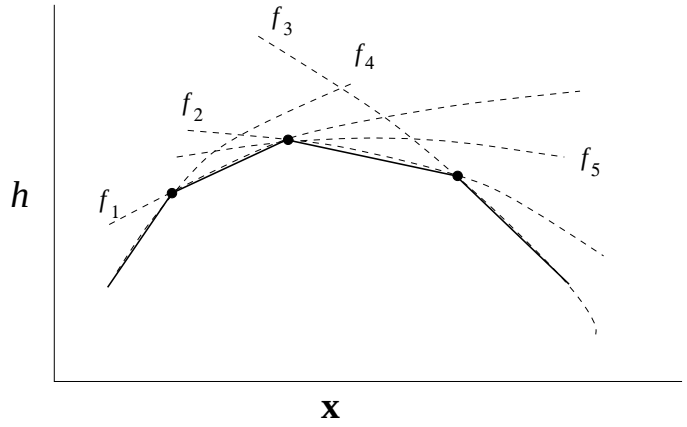

Figure 5.4: Cross-section of the objective function in an optimization-based mesh-smoothing problem

Smoothing drivers are found in `src/2D/MeshOpt2D.cxx` and `src/3D/MeshOpt3D.cxx`. The OptMS library is in `src/smoothing`.

5.3 Default mesh optimization in GRUMMP

In two dimensions, the meshes GRUMMP generates in the `tri` executable are already excellent. The `meshopt2d` executable actually operates by computing an appropriate length scale for the input mesh, then using Ruppert's scheme to refine the mesh until quality is high; the guarantees on cell size provided with size and grading control in [11] ensure that the output mesh is still of reasonable size. In both cases, post-processing by using a pass or two smoothing to maximize the minimum sine of angles in the mesh generally increases the minimum angle by a few degrees at low cost.

In three dimensions, GRUMMP again produces good meshes in the `tetra` executable. Because slivers are still generated, `tetra` post-processes its meshes by swapping to increase the minimum sine of the dihedral angles in the mesh, and performs one pass of smoothing with the same goal. `meshopt3d` differs from its 2D counterpart in that insertion is not part of its operation; instead, a user-definable sequence of swapping, smoothing, and bad tetrahedron repair steps are performed. The default optimization sequence follows the experimentally-inspired recommendations of Freitag and Ollivier-Gooch [3], which typically performs well for meshes that are approximately isotropic.

Chapter 6

Mesh Coarsening

GRUMMP provides mesh coarsening capability to enable automatic generation of quality coarse meshes for use with multigrid. The algorithm [10] selects a set of vertices from the fine mesh to retain in the coarse mesh, then removes unwanted vertices one at a time from the fine mesh. This incremental deletion scheme has a robustness advantage over generating a coarse mesh from scratch: incremental deletion *always* produces a coarse mesh even if it can not remove all the unwanted vertices. The scheme has been designed to coarsen triangulations by a factor of about 2^D in D dimensions for each coarsening while preserving important geometric and topological features of the fine mesh and producing coarse mesh cells with acceptable shape quality.

6.1 Selecting Vertices to Retain in the Coarse Mesh

An experienced analyst can look at an unstructured mesh and identify important features that should be retained in a coarse mesh. For example, a person would choose to retain the vertex at the sharp trailing edge of an airfoil in two dimensions and corners in three-dimensional geometry. Elsewhere in the mesh, a person would choose a sampling of points, not too close together but also not leaving any large blank areas. This section describes a vertex selection algorithm that makes the same choices. Critical to the success of the algorithm is the notion of a hierarchy of features in the mesh, from sharp corners on the boundary to plain interior vertices. The algorithm is summarized below in terms of this hierarchy, from most to least specialized; the remainder of this section discusses the details of the algorithm.

1. An *apex* is a boundary vertex at which a sharp corner is formed; examples are identified in Figure 6.1 by solid circles. Apexes are always included in the coarse mesh.
2. A *fold* is a line on the surface of a three-dimensional object where the surface normal is discontinuous.[1] A typical example for aerospace applications is the trailing edge of a wing, as shown schematically in Figure 6.1 (bold lines). For isotropic surface meshes, alternate fold vertices are retained.

   Identification of apexes and folds is discussed in Section 6.1.1.

[1] In practice, a user-defined angle is used as the criterion for discontinuity.

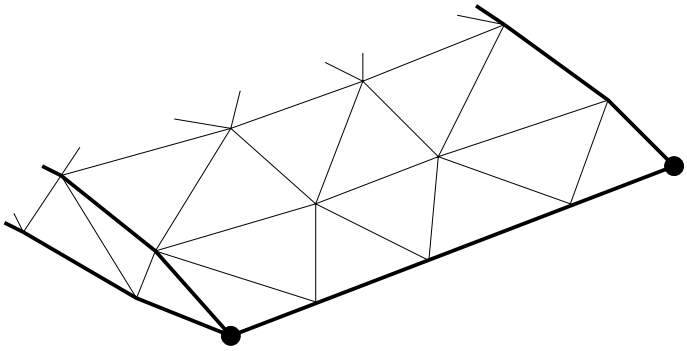

Figure 6.1: Examples of apexes and folds.

3. A *maximal independent set* (MIS) of the remaining boundary vertices is included in the coarse mesh. That is, a set of fine mesh vertices are chosen such that the included vertices are not too geometrically close to each other *and* that every excluded vertex *is* too close to at least one included vertex. See Section 6.1.2 for details.

4. Finally, an MIS of the remaining interior vertices is selected for inclusion in the coarse mesh.

6.1.1 Identification of Apexes and Folds

Apexes and folds are lower-dimensional mesh features: an apex is a zero-dimensional entity, while a fold is a one-dimensional entity in a three-dimensional mesh. To determine whether a boundary vertex is an apex or lies on a fold, the algorithm computes the unit normals of all faces (edges in 2D, triangles in 3D) that are incident on the vertex.

In two dimensions, only two faces are incident on a vertex. If these faces have normals that differ in direction by more than 20^o, the vertex is classified as an apex.

In three dimensions, the presence of two distinct face normals (separated by some user-defined angle β) generally means that the vertex lies on a fold, whereas three distinct normals are present if and only if the vertex is an apex. Three cautionary notes are in order here.

1. The angle difference at which two normals are considered "distinct" must be chosen with attention to surface curvature. If this angle is chosen too small, then many edges in the mesh of a curved surface will be improperly labeled as folds. Advanced techniques in surface reconstruction and curvature estimation (see, for example, [6, 13]) may be of substantial benefit in distinguishing between a coarse surface mesh and an abrupt change in the underlying smooth surface; adding such capability to GRUMMP is planned.

2. In identifying apexes, it is not enough to compare only the normals of adjacent surface faces. A sharp-tipped cone can easily be constructed with enough surface faces incident on its apex that their normals differ in direction by an arbitrarily small amount. Instead, we must group faces so that members of each group have normals that differ by less than β. The number of such *groups* is the deciding factor in whether a vertex is an apex.

3. A vertex can have only two groups of normals and still be an apex. As an example, suppose that the two visible faces in the lower-left corner of Figure 6.1 were nearly coplanar. To detect this case, the algorithm checks for near-collinearity of the edges that separate the two groups. If the edges are nearly collinear, the vertex lies on a fold; otherwise, it is an apex.

6.1.2 Selection of Points to Include in the Coarse Mesh

Apexes are always retained in the coarse mesh. This choice has positive and negative side effects. On the positive side, not object can ever be eliminated from the mesh if apexes are retained. On the negative side, small features on large objects are likely to be retained, even when typical cell sizes are much larger than the feature in question.

Elsewhere, the mesh is coarsened with the goal of reducing mesh size while maintaining a good distribution of vertices and therefore good mesh quality. One intuitive approach to this problem is to keep as many vertices as possible without keeping two that are too close together: a maximal independent set (MIS) of the conflict graph of the mesh [7]. In the conflict graph, an edge connects any pair of vertices that are physically too close together compared with the length scale defined for those two vertices. This graph is similar but not identical to the connectivity graph, as occasionally first neighbor vertices in the mesh are far enough apart to co-exist in the coarse mesh, whereas some second neighbor vertices are too close together.

For efficiency reasons, GRUMMP constructs the conflict graph as needed. Before using the MIS to coarsen boundary curves, the conflict graph for boundary curve vertices is constructed. Later, when the MIS for boundary vertices is needed, that conflict graph is constructed, and likewise for isotropic interior vertices. Because only currently active vertices have their conflict graph computed at each step, no vertex ever has its conflict graph re-generated.

GRUMMP uses MIS vertex selection in three places: folds, surface meshes, and interior meshes. Applying the same MIS code for these three cases requires only appropriate restrictions on which vertices are *active*, as these are the only vertices that the algorithm can legally mark for inclusion or exclusion from the coarse mesh. For selection of surface vertices, for example, only plain surface vertices are active; any specialized surface vertices (apexes or folds) and all interior vertices are inactive. In each case, the maximal independent vertex set is constructed in two phases: initial creation and size improvement. In the initial creation phase, the algorithm traverses the mesh using an advancing front, marking any unmarked, active vertex v_A for inclusion in the coarse mesh and all active vertices which conflict with v_A for exclusion. Next, several passes are made to increase the size of the MIS, with the expectation that this will make edge lengths match the intended mesh length scale more closely in the coarse mesh.

6.2 Incremental Vertex Deletion

Incremental vertex deletion begins with an existing valid fine mesh and removes unwanted vertices one at a time, with a valid mesh after each deletion. This approach always results in a valid coarse mesh, even though it does not necessarily remove all unwanted vertices. In two dimensions, vertex deletion succeeds in removing all vertices for cases to date, although there is no proof of this property. In three dimensions, vertex deletion often leaves a few unwanted vertices in the coarse mesh, but always a small fraction of requested deletions.

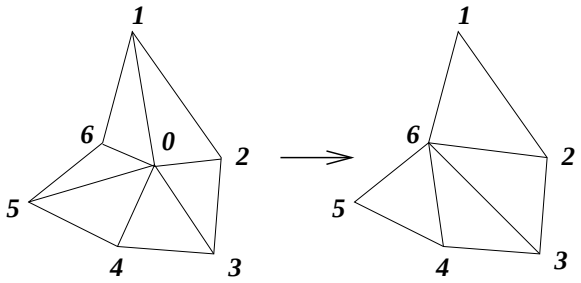

Figure 6.2: Vertex removal by edge contraction in two dimensions

The kernel of the incremental vertex deletion algorithm is an edge-contraction algorithm for deleting a single vertex.[2] A vertex is removed by shrinking one of its incident edges to zero length, as illustrated in two dimensions in Figure 6.2. The left half of the figure shows a vertex 0 — which is to be removed from the mesh — and its immediate neighbors in the mesh. Vertex 0 will be removed by sliding it along the edge $\overline{06}$ to vertex 6. In the process, cells $\triangle 061$ and $\triangle 056$ are removed, as are edges $\overline{01}$, $\overline{05}$ and $\overline{06}$. The resulting mesh fragment is shown in the right half of Figure 6.2. The kernel extends directly to three dimensions, with additional bookkeeping as the only complication.

6.2.1 Invoking Vertex Deletion

The incremental deletion algorithm makes multiple passes through the mesh, continuing until a pass removes no vertices or until all vertices selected for deletion have been removed. Between passes, local reconnection and smoothing are applied to improve the poorest-quality regions of the mesh and facilitate further vertex removal. During each pass, for each vertex marked for deletion in the coarse mesh, an edge is selected for contraction based on the following criteria:

1. Feature hierarchy. A vertex must be removed by contraction onto a vertex that is no lower in the feature hierarchy. For example, surface vertices must be moved onto other surface vertices (including fold and apex vertices). Without this restriction, a surface vertex could be removed by contraction onto an interior vertex, changing the surface shape significantly. This rule also prevents saw-toothing of folds in three-dimensional meshes.

2. Mesh validity. The resulting mesh fragment must not have inverted cells. For example, in Figure 6.2, vertex 0 can not be moved onto vertex 1 or vertex 5 without creating a cell ($\triangle 156$) with a negative area.

3. Mesh quality. The algorithm chooses the edge contraction that gives the best quality for the resulting mesh fragment. A good choice of quality measure here is the maxmin sine criterion. This criterion maximizes the minimum sine of the angles of the cell (dihedral angles in 3D), avoiding both large and small angles. For the case in Figure 6.2, this rules out moving vertex 0 onto vertex 3, because this contraction produces the poorly-shaped cell $\triangle 345$. If edge contraction would form too poor a cell in three dimensions, the vertex is not removed; often the same vertex *can* be successfully removed in a subsequent pass through the mesh. This cutoff is unnecessary in two dimensions.

[2] A preliminary version of this algorithm has been described elsewhere [8].

6.2.2 Mesh Post-processing

After the incremental deletion algorithm has made a complete pass through the mesh, local reconnection is used to improve mesh connectivity, and therefore the effectiveness of edge contraction in the next pass. In this context, we reconnect the mesh using both face and edge swapping and perform one pass of optimization-based smoothing with a floating threshold [3]. Both mesh optimization techniques have the goal of maximizing the minimum sine of angles.

After all vertex removal is complete, we do additional post-processing to improve coarse mesh quality. In two dimensions, we use five passes of optimization-based smoothing with a floating threshold, using the maxmin sine criterion. After the third pass, we use a single pass of mesh reconnection. In three dimensions, we again use a combination of smoothing and swapping, beginning with two passes of optimization-based smoothing with a floating threshold, followed by application of heuristics aimed at removing the worst cells in the mesh by using face and edge swapping, and finishing with two more passes of smoothing. Our post-processing regimen for three-dimensional meshes is consistent with the experimentally-inspired recommendations of Freitag and Ollivier-Gooch [3]. Post-processing improves the smallest angle in the output mesh significantly and cost is quite modest, at less than 10% of CPU time in two dimensions and far less than that in three dimensions.

6.3 Anisotropic Mesh Coarsening

For anisotropic problems — including notably external viscous flow problems — there is often a pseudo-structured part of the mesh to accurately resolve anisotropic physics. The pseudo-structured part of the mesh typically contains quadrilaterals, prisms, or hexahedra that have been divided into triangles or tetrahedra. Pseudo-structured meshes are useful in treatment of anisotropic physics, and these pseudo-structured regions should be preserved in coarse meshes to the extent feasible.

These pseudo-structured mesh regions can be coarsened isotropically, reducing the number of vertices by a factor of 2^D in these regions by removing alternate planes of points in each direction. However, some work [5, 12] suggests that multigrid methods are much more efficient if coarsening is done anisotropically, reducing the cell aspect ratio near the wall and the associated numerical stiffness. We allow several variations on anisotropic coarsening to accommodate different scenarios for surface and interior mesh pseudo-structure, as discussed in Section 6.3.1.

1. Three-dimensional meshes may have a locally anisotropic, *pseudo-structured surface mesh*, as shown in Figure 6.3. Roughly speaking, in this case, all fold vertices are retained, and alternate vertices are retained along closely-spaced lines leaving the fold. This process is described in detail in Section 6.3.1.

2. Both two- and three-dimensional meshes may have sections of locally anisotropic, *pseudo-structured interior mesh*, similar in two dimensions to the example in Figure 6.3. Pseudo-structured interior mesh fragments are coarsened in much the same way as pseudo-structured surface mesh fragments.

6.3.1 Selection of Points in Pseudo-structured Anisotropic Meshes

In selecting points with pseudo-structured anisotropic mesh fragments (PSAMF's) to retain in coarse meshes, we consider three important and distinct cases:

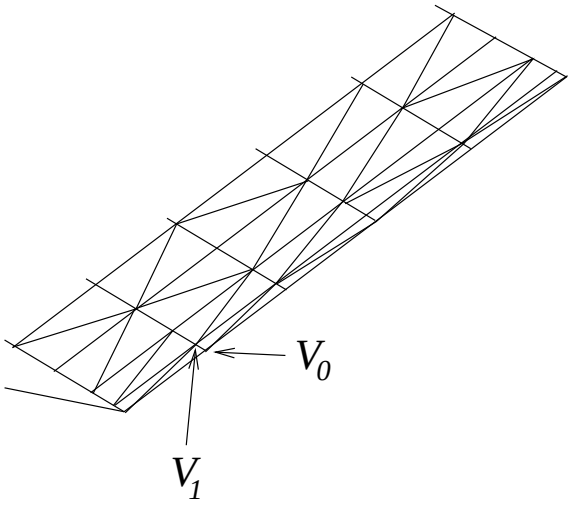

Figure 6.3: Pseudo-structured surface mesh on a wedge.

1. The fully-anisotropic case (3D). Some three dimensional meshes have anisotropic layers built out from *anisotropic* surface triangulations. A typical example of this is found in computational aerodynamics, where cells near aircraft wings for viscous simulations are much larger in the spanwise direction than in the streamwise direction, and much larger in the streamwise direction than normal to the wing surface. To reach unit aspect ratio in both directions during coarsening, we initially keep every point along the trailing edge in the spanwise direction, alternate lines on the wing surface, and every fourth layer in the interior of the mesh. This reduces the aspect ratio in both directions, and reduces the number of vertices by a factor of 8 in these mesh fragments. As the spanwise-streamwise aspect ratio approaches one (*i.e.*, as the surface mesh becomes isotropic), such a mesh fragment would revert to Case 2.

2. The semi-anisotropic case (3D). Some three dimensional meshes have anisotropic layers built out from *isotropic* surface triangulations. In this case, we choose to coarsen the surface mesh as we would a two-dimensional isotropic mesh. In the interior, we choose to emphasize reduced cell aspect ratio near the boundary by selecting every fourth vertex along marching lines beginning at the surface vertices that will be retained. This approach reduces the aspect ratio of anisotropic cells near the boundary by roughly a factor of two at each coarsening and reduces the number of vertices in such mesh fragments by a factor of 16.

3. In two dimensions, directional anisotropic coarsening should select every fourth vertex along marching lines extending from each point on the boundary into the domain.

These rules are simple to describe and simple for a person to follow, but some care is required so that a computer can reliably identify a pseudo-structured anisotropic mesh fragment and coarsen it properly. For details, see [9].

6.4 Outcomes

GRUMMP is quite successful in coarsening isotropic unstructured meshes — nearly all vertices are removed in each case, and mesh quality is quite good. Two-dimensional anisotropic meshes are also handled well. The weakest area, by far, is anisotropic three-dimensional meshes, where the number of vertices that can not be removed climbs sharply, and mesh quality is poor (even allowing for anisotropy).

Two strategies that have some chance at success have not yet been investigated:

- Local transformation of anisotropic mesh fragments into a space where the mesh is approximately isotropic. In this scenario, edge contraction could be applied in the transformed space. The isotropic edge contraction criteria should strongly inhibit tangling of connectivity among vertices that should be in different levels of the anisotropic mesh.
- Creation of the coarse anisotropic mesh by construction. This approach would likely involve determining the coarse surface mesh topology and then projecting that topology out through the appropriate number of layers of anisotropic cells. A major challenge with this approach would be transition from anisotropic to isotropic meshing. Also, this approach is tantamount to generating the coarse mesh from scratch; the latter might give more flexibility and therefore be preferable for anisotropic meshes.

Chapter 7

Isotropic Mesh Adaptation

GRUMMP's approach to isotropic mesh adaptation is to apply a concatenation of existing functionality to produce a high-quality mesh that meets the specified length scale requirement. (See Section 2.1 for information on how to specify a length scale.) This is done as the first step in mesh optimization (executables `meshopt2d` and `meshopt3d`). The steps taken are:

1. Coarsen the mesh where necessary to make sure vertices are not too close together (where "too close" means that the distance between them is less than the average of their length scales).
2. Swap the mesh to make it Delaunay; this step is a necessary prerequisite to mesh refinement.
3. Refine the mesh to ensure that no cells are larger than the local length scale and that no badly-shaped cells remain. This step is identical to what GRUMMP does in generating a mesh after the initial triangulation / tetrahedralization is complete.
4. Postprocessing by smoothing and swapping. In 2D, two passes of smoothing are sufficient to ensure excellent meshes in both theory and practice. In 3D, some swapping is typically required to eliminate slivers, which the insertion process alone cannot remove.

If this sounds simple, that's because it is: the driver for this process has eight lines of code, not counting diagnostic output, comments, and blank lines.

Bibliography

[1] Charles Boivin and Carl F. Ollivier-Gooch. Guaranteed-quality triangular mesh generation for domains with curved boundaries. *International Journal for Numerical Methods in Engineering*, 55(10):1185–1213, December 2002.

[2] Lori Freitag, Mark Jones, and Paul Plassmann. An efficient parallel algorithm for mesh smoothing. In *Proceedings of the Fourth International Meshing Roundtable*, pages 47–58. Sandia National Laboratories, October 1995.

[3] Lori A. Freitag and Carl F. Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21):3979–4002, 1997.

[4] David L. Marcum and Nigel P. Weatherill. Unstructured grid generation using iterative point insertion and local reconnection. AIAA paper 94-1926, January 1994.

[5] Dimitri J. Mavriplis. Multigrid strategies for viscous flow solvers on anisotropic unstructured meshes. In *Proceedings of the Thirteenth AIAA Computational Fluid Dynamics Conference*, pages 659–675. American Institute of Aeronautics and Astronautics, July 1997. AIAA 97-1952-CP.

[6] Mark Meyer, Matthieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. Presented at the International Workshop on Visualization and Mathematics 2002, Berlin, Germany, May 2002.

[7] Gary L. Miller, Dafna Talmor, and Shang-Hua Teng. Optimal coarsening of unstructured meshes. *Journal of Algorithms*, 31(1):29–65, 1999.

[8] Carl F. Ollivier-Gooch. Multigrid acceleration of an upwind Euler solver on unstructured meshes. *American Institute of Aeronautics and Astronautics Journal*, 33(10):1822–1827, October 1995.

[9] Carl F. Ollivier-Gooch. Robust coarsening of unstructured meshes for multigrid methods. In *Proceedings of the Fourteenth AIAA Computational Fluid Dynamics Conference*, pages 1–8. American Institute of Aeronautics and Astronautics, July 1999.

[10] Carl F. Ollivier-Gooch. Coarsening unstructured meshes by edge contraction. *International Journal for Numerical Methods in Engineering*, 57(3):391–414, May 2003.

[11] Carl F. Ollivier-Gooch and Charles Boivin. Guaranteed-quality simplicial mesh generation with cell size and grading control. *Engineering with Computers*, 17(3):269–286, 2001.

[12] Niles Pierce, Michael Giles, Antony Jameson, and Luigi Martinelli. Accelerating three-dimensional Navier-Stokes calculations. In *Proceedings of the Thirteenth AIAA Computational Fluid Dynamics Conference*, pages 676–698. American Institute of Aeronautics and Astronautics, July 1997. AIAA 97-1953-CP.

[13] A. Rassineux, P. Villon, J.-M. Savignat, and O. Stab. Surface remeshing by local hermite diffuse interpolation. *International Journal for Numerical Methods in Engineering*, 49:31–49, 2000.

[14] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Proceedings of the Fourth ACM-SIAM Symposium on Discrete Algorithms*, pages 83–92, 1993.

[15] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, May 1995.

[16] Jonathan R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.

[17] David F. Watson. Computing the *n*-dimensional Delaunay tessellation with application to Voronoi polytopes. *Computer Journal*, 24(2):167–172, 1981.

Appendix A

Changes Between Versions

A.1 From version 0.6.3 to 0.6.4:

- Version 0.6.3 is an ITAPS maintainence release, for compatibility with version 1.3 of the iMesh API.

A.2 From version 0.6.2 to 0.6.3:

- Significant improvements in surface recovery for polyhedral boundary data. Not only is recovery more robust, but it also now leaves the surface triangulation untouched. That is, the surface triangles of the initial volume mesh now match those of the input boundary. Polygons in the input data are triangulated as usual.
- Added support for the Observer paradigm in mesh classes. These classes now publish changes made to the mesh database to an observers that request them. This is done in batch (blocks of 100, currently) to reduce message overhead. This will allow us — in time — to fully unwind the interactions between various mesh modification algorithms that are currently mediated through flags on entities in the mesh database.
- Begun the process of factoring algorithms (swapping, smoothing, insertion, etc) out of the mesh database classes. At this point, swapping has made the most progress, with a fully functional external swap manager, which uses the Observer capabilities to keep track of which faces and edges should be considered for swapping. There are still a handful of places where swapping is still done the "old way", using mesh member functions, so that code, though deprecated, still exists.
- Upgraded to CGM 10.2.3.
- Release 0.6.3 is compatible with version 1.2 of the ITAPS mesh API (iMesh).

A.3 From version 0.6.0 to 0.6.1/0.6.2:

Minor ITAPS-related cleanup; no other changes.

A.4 From version 0.5.0 to 0.6.0:

Version 0.6.0 brings another significant upgrade to GRUMMP, which now supports meshing from curved boundary data. Starting from the curved boundary representation, GRUMMP discretizes surfaces by sampling, then generates tetrahedral meshes from the resulting surface triangulations. Any boundary points that are inserted during volume refinement are inserted onto the curved surface, not onto a tesselation of the surface. Geometry is currently input via stereolithography files (see Section 3.3 for more information). As a consequence of how geometry is handled internally, GRUMMP now requires the Argonne version of the Common Geometry Module (version 10.2.2) to build. The configure script automatically fetches and builds this if it's needed.[1]

Additional changes since version 0.5.0:

- Fixed numerous errors and warnings when compiling with gcc 4.4. This is what we get for making a release with an older compiler.
- Support for reading legacy ASCII VTK files, binary UGrid, and binary VGrid files in 3D; these are auto-detected at read time. This is in addition to GRUMMP's template approach for auto-generating code for reading ASCII mesh files.
- Infrastructure changes to centralize manipulation of the mesh topology database. These changes aren't user-visible: the same things happen at almost exactly the same speed. But with this change to better encapsulate mesh topology changes, we expect to be able to relatively easily add support for edges in the 3D mesh database, so that we can generate curved meshes.
- Various other bug fixes.

A.5 From version 0.3.3 to 0.5.0:

Version 0.5.0 brings a significant upgrade to GRUMMP, with the addition of support for the ITAPS mesh interface. This interface provides mesh query and basic modification functionality in a data-structure neutral way, as well as supporting collections of mesh entities (sets) and arbitrary application-defined data on mesh entities and sets (tags). Of particular interest to Fortran users who might be interested in using the GRUMMP mesh database but reluctant to switch to C++, the API can be accessed from Fortran (77 through 2003).

Additional changes since version 0.3.3:

[1] As of version 0.6.0beta1, meshing from STL files is temporarily disabled, as parts of its internals are incompatible with the infrastructure changes below; the final 0.6.0 release will address this.

- Fixed an egregious error in 3D initial tetrahedralization, which had the unfortunate result that many input files failed to generate meshes at all. (Should have been a 0.3.4 at some point to fix this, but somehow that never got done...)
- Improved treatment of curved boundaries in 2D; these changes have little user-visible effect other than producing somewhat smaller meshes for some inputs.
- Began migration towards having specific reader/writer routines for common file formats. Presently, this is in addition to the current custom ASCII format capability; the latter may eventually be replaced by some form of plug-in architecture, although there are currently no definite plans on this point.
  - Support for reading 3D meshes generated with AFLR3D and VGRID. Also, support for reading standard FEA (element-vertex) mesh connectivity in 3D.
  - Direct support for writing 2D meshes in FEA format, including high-order curved boundary information.
  - Input now checks to see whether a file extension is present rather than automatically adding one. A long-standing annoyance with a five-minute fix...
- Internal modifications to the queueing system used to prioritize mesh entities for point insertion; not user visible.

A.6 From version 0.3.3 to 0.4.0:

Version 0.4.0 is a special-purpose, limited release for a UBC user needing periodic meshing in 2D.

- **Added partial support for periodic meshing in 2D.** Any two linear boundary patches can be identified as being periodic with each other. A geometric mapping between the two is automatically determined, and when a vertex is inserted on one periodic boundary, it is automatically created on the other as well. This capability is not likely, at this point, to be robust in the presence of small input angles or input points near (i.e., encroaching on) the periodic boundary.

A.7 From version 0.3.2 to 0.3.3:

Version 0.3.3 is a bugfix release, including fixes for:

- Fixed a number of minor bugs, including
  - 3D patch optimization (thanks to Nigel Nunn for finding the bug and providing a patch).
  - Pointer arithmetic in the variable-sized array pseudo-template. This bug caused problems only with gcc4, because other compilers generate incorrect code that still miraculously ran correctly; gcc4 removed this serendipitous fix.
  - A string problem in termination of GRUMMP executables.

  - A crash in `scat3d` when extrapolating data.
  - Can now use `index` in boundary face and internal boundary face descriptors in the I/O generator.
  - Orientation fix for certain multiregion input files in 3D.
  - Supply a default BC properly in 2D when reading a mesh without BC's.

- Replaced MAX and MIN macros with std::max and std::min, because gcc4 complains that the macros are deprecated, even though the GRUMMP headers defined them explicitly.

A.8 From version 0.3.1 to 0.3.2:

Version 0.3.2 is a bugfix release.

- **More minor updates for TSTT.** Added VTK input files to distribution tarball. Recognize a TSTT configure option (–enable-tsttm) as a sign that this build is intended for use with TSTT, to auto-disable logging. Added a 'make all' target in the top-level Makefile.

A.9 From version 0.3.0 to 0.3.1:

Version 0.3.1 is a bugfix release.

- **Scattered data interpolation.** In both 2D and 3D, fixed bugs that caused seg faults when trying to do scattered data interpolation.
- **Minor updates for TSTT.** Added a configure option for VTK-format input into GRUMMP. Fixed several compile errors when logging is disabled. Also tweaked a couple of things in the main Makefile.
- **Length scale reporting.** A bug in 0.3.0 made it so that the edge length scale info reported after mesh re-ordering was incorrect; this has been fixed.

A.10 From version 0.2.1 to 0.3.0:

- **Isotropic mesh adaption.** The biggest user-visible change since the previous release is support for isotropic mesh adaptation. Internally, GRUMMP has had this functionality for a while, but by user request, it's now accessible through the command line interface. Basically, length scale information can be supplied for some or all vertices in a separate file, and meshopt2d / meshopt3d will modify the input mesh to match that length scale. If no length scale is provided, meshopt2d/3d behave as they always have.
- **Length scale improvements.** Fixed a problem with length scale calculations that sometimes caused a vertex to identify a bdry patch that was just -behind- the vertex (and therefore not visible) as one of its nearby patches. The 3D tire incinerator example was the geometry that showed this problem the most distinctly.

- **Improved initial tetrahedralization.** More incremental improvement to initial tetrahedralization, most notably in merging co-planar input facets and better handling of small angles between input edges. The latter also has the effect of reducing output mesh size noticeably for many input files with small angles between edges.
  - Added support for merging co-planar boundary patches in 3D. For input files that have large numbers of co-planar triangles, this boundary optimization can be a tremendous help in creating the initial volume mesh. All vertices are guaranteed to be in the volume mesh, but there are no guarantees about the surface triangles. (Command line option: -m for merge bdry patches.)
  - Fixed a bug in handling surface recovery for multi-region bdry files in 3D. This bug often resulted in failures near the end of initial tetrahedralization.
  - Support for free vertices in input files for 3D meshes. That is, a vertex can be pre-defined to have a particular location -without- being connected to the surface mesh. (The point must lie on or inside the domain, obviously.) Because of differences in the way that 2D and 3D input is handled, this capability is tricky to implement in 2D. However, a currently-underway re-write of all boundary input handling should enable this in 2D before much longer.
- **Reordering of mesh entities** before output to improve bandwidth and cache performance for solvers reading these meshes. Vertices are reordered using the reverse Cuthill-McKee algorithm. Faces and cells are reordered according to the sum of the indices of their vertices.
- **Support for long arcs**: circular arcs longer than 180 degrees. An "arc" is still CCW and < 180 degrees. A "longarc" is CCW and > 180 degrees between the same two points.
- Fixed a **minor edge swapping bug** that often resulted in a neglecting a few swaps.
- Internal changes to **support the Terascale Simulation Tools and Technologies (TSTT) mesh interface**. This is a language- and data-structure neutral interface to meshing tools. For more information, including links to the TSTT interface definition, see http://www.tstt-scidac.org.
- Fixed a GNU/Linux shared library creation bug. Added support for shared libraries under AIX (finally!).
- Made changes for GCC 3.x compatibility (mostly tightening up use of std::) and to improve portability (mostly getting rid of uses of the GNU extension that allows dynamic arrays to be declared as name[run-time size].
- Removed some experimental and obsolete files from the distribution.

A.11 From version 0.2.0. to 0.2.1:

- **Improved mesh coarsening**. This is by far the biggest change between versions. Both 2D and 3D coarsening are affected. Coarse mesh quality for isotropic meshes has improved significantly, as a result of changing the way coarse mesh vertices are selected. Vertex removal efficiency is now better in 3D, with typically fewer than one vertex in a thousand *not* removed as requested. Anisotropic coarsening in 2D working very well; in 3D, anisotropic coarsening is miserably poor because of challenges with mesh connectivity.

- **Corrected a series of small errors in mesh quality assessment**. Minimum solid angle and the aspect ratio measures now work in 3D. Also, ratio of incircle to circumcircle radius is now normalized properly in 2D.
- **Internal changes in how smoothing is handled.** These changes were made to allow solvers to call GRUMMP meshing routines, including smoothing, without needing to know how GRUMMP handles things internally.
- **Re-introduced the "precious boundary" flag.** By popular demand, the flag that tells `tetra` not to modify the mesh boundary has been added back to the code. Use this flag only in duress, because:
  - Initial tetrahedralization still adds points to the boundary when it needs to.
  - There are no mesh quality guarantees, or even practical expectations, unless points can be inserted on the boundary. Things may work out for you, and they may not.
- **Slightly improved diagnostics and on-the-fly repair for bad input files**. Some 3D input files with mis-oriented triangles can now be fixed. Others are identified as bad input files. Unfortunately, some can still escape detection and cause code crashes after surface recovery.
- Documentation now includes overview of algorithms used in GRUMMP.
- Fixed a couple of egregious bugs in the rarely-exercised code for reading 3D meshes in cell-vertex (FEM) data structure.
- Fixed a small but annoying bug that made new `gcc` versions fail to compile.

A.12 From version 0.1.7 to 0.2.0:

- **Improved robustness in initial tetrahedralization**. A few cases still fail, but changes to the heuristics used in surface recovery break other cases. An area of vast improvement over version 0.1.7, but more work is still needed.
- Support for **curved boundaries in two dimensions**. Circles, circular arcs, and splines are supported, as well as straight line segments. See the documentation for `tri` for details about file format.
- **Guaranteed-quality meshing**. In 2D, GRUMMP now generates meshes with all angles greater than 30°, except near input angles smaller than about 60°. In 3D, the theoretical guarantee is on the ratio of edge length to circumradius; in practice, with smoothing GRUMMP produces 3D meshes for "thick" domains with dihedral angles in excess of 18°-20° (except, again, near small angles in the input). A bonus with these new algorithms is that mesh generation is actually *faster* than in version 0.1.7. (It hardly seems fair that such a massive change in the internals of the code lead to only one item in the change list...)
- **Improved length scale calculation**. GRUMMP now uses, in both 2D and 3D, a much better approximation to the true local geometric feature size.
- **Global control of cell size** relative to feature size and of the rate of change of cell size (grading). Both of these accompany the improved length scale calculation.

- **Change in output format for multi-domain meshes**. Because of internal changes in data representation, output file format has been changed for multi-domain meshes. As a result, two default file format templates now exist, one each for single- and multi-domain cases.
- **Internal cleanup continues**. Quite a bit of obsolete code has been deprecated, and many of these pieces have been removed from the distribution.

A.13 From version 0.1.6a to 0.1.7:

- Support for **multi-domain meshing**. That is, it is now possible to specify a problem domain that must be meshed as two or more (max: 31) adjacent regions, with the boundaries between them meshed consistently. This is the biggest addition to this version of GRUMMP.
- **Improved boundary data input format**. This was necessary to support multi-domain meshing, and the new format was designed to be extensible, with an eye towards meshing from non-polygonal/polyhedral data.
- **Improved geometric calculations**. GRUMMP often needs to know the orientation of $N+1$ points in N dimensions or whether a given point is inside the ball formed by $N+1$ other points. Earlier versions used simple floating point arithmetic to determine these things, and sometimes ran into trouble as a result. This version incorporates adaptive-precision versions of these predicates that were developed by Jonathon Shewchuk of U California, Berkeley, while he was a PhD student at Carnegie Mellon University.
- **File name changes**. All GRUMMP include files[2] are now prefixed with `GR_` to avoid naming conflicts. Also, in deference to non-case-sensitive operating systems, all C++ files now have a `.cxx` suffix instead of `.C`.
- **Minor bug fixes**. All but a couple of the problem input files sent to me after version 0.1.6 now work.
- **Changes to quote handling for I/O generation.** The old way of handling non-keyword text in I/O template files made it impossible to have

  ```
  x 0.5 y 0.7
  ```

  as a vertex coordinate line; the ‘x’ and ‘y’ were not handled properly. These changes affect all cases where quotes are used in template files, so check any template files you have created to make sure they are compatible with the new form of quote handling.
- **Unified `lex` input files for I/O generation**. All input generation is now done from a single `lex` file, and all output generation is done from a (different) single `lex` file. This removed inconsistencies between the 2D and 3D versions of the I/O generation code.

[2] Except those for the logging and smoothing libraries, which have their own name space.

A.14 From version 0.1.6 to 0.1.6a:

- Support for **HP-UX 10.20 with native compilers**. This required a large number of small changes, with three common causes.

  1. The HP-UX C++ compiler has a broken implementation of the **inline** keyword: it chokes on inline functions with loops instead of implementing them out-of-line.
  2. The HP-UX C++ compiler does link-time template instantiation.
  3. A number of warning messages, for both C and C++ code, gave useful pointers to idioms in the code that were worth tightening up.

  This cleanup should also improve portability to other new platforms, especially those with `cfront`-based compilers.

  Thanks to Aldo Bonfiglioli in the Faculty of Engineering at the University of Basilicata for providing accessing to an HP-UX 10.20 test machine.

- Minor improvements to the `lex` input files for I/O generation.

A.15 From version 0.1.5 to 0.1.6:

- **Three-dimensional coarse mesh generation** for multigrid. In principle, this could be done by running `meshopt3d -s 2`. In practice, there is a useful optimization that allows rapid removal of nearly all of the extra vertices from the coarse mesh. `coarsen3d` does this, and then behaves just like `meshopt3d`.
- **Two-dimensional scattered data interpolation.** A new executable, `scat2d`, has been added for this.
- **Default optimization parameter in 3D** is now to swap and smooth, rather than using the much more expensive refine-to-length scale.
- **Improved quality** of initially-generated meshes in 3D.
- **Improved speed** for refine-to-length in 3D.
- **HTML documentation added**, both on the GRUMMP web page and in the doc/html directory in the distribution.
- **Two new quality measures** in 2D. These are actually measures that were already advertised — ratio of incircle to circumcircle radius, and ratio of area to perimeter squared. However, in version 0.1.5 and previously, these measures were not implemented properly.
- **Surface vertex removal** in 3D. There are a number of special cases that had to be implemented before this was safe.
- **Unified smoothing calls in 3D.** In version 0.1.5, surface smoothing was enabled only when smoothing all vertices in the mesh. With the consolidation of some duplicate code, this problem has been fixed.

- **Internal code cleanup and optimization.**
- **Made logging more systematic.**

A.16 From version 0.1.4 to 0.1.5:

- **Surface smoothing** implemented for points on flat surfaces (2D and 3D).
- **Reconfiguration of non-planar surface edges** allowed in 3D up to a user-specified tolerance on non-planarity.
- **Length-scale calculation in 3D** now much improved. Previously, this calculation was quite poor in some circumstances. More work still needs to be done here.
- **Unified source** for `tri`/`meshopt2d` and `tetra`/`meshopt3d`.
- **Internal code clean up**, including some additional state info that will soon improve point deletion, especially in 3D.
- **Streamlined general mesh improvement algorithm** via better decision-making about which points to try to smooth and which faces to try to swap in mesh improvement. Some speed improvement as a result, but intrinsically slow.
- Mesh improvement with approximately **constant total mesh size**.
- Better documentation for **shared library use with sh-type shells**.
- Documentation of command-line **mesh optimization strings**.
- Attempt to improve **compatibility with HP-UX** 10.x. There have been reports of some difficulties with building GRUMMP on this platform. This is not a well-tested attempt at fixing the problem, given my lack of an HP-UX 10.x test platform; if you try GRUMMP on HP-UX 10.x with the native compilers, let me know the results, good or bad.

A.17 From version 0.1.3 to 0.1.4:

- **Fixed major** but subtle **bug** in internal `List` handling **that caused segmentation faults** (bug introduced during cleanup for 0.1.3).
- **Fixed crash** when specifying `-e [01]` in `tetra` and `meshopt3d`.
- **Minor cosmetic fix** in `configure`.

A.18 From version 0.1.2 to 0.1.3:

- Fixed bug in which input **boundary conditions were not transmitted** to the output volume mesh in three dimensions.
- Fixed bug in which some **input polygons were triangulated incorrectly** due to a sorting ambiguity on some machines (including IRIX and Solaris).
- **Corrected several erroneous input test files**.
- **Modified interface for `scat3d`**. Also, `scat3d` will now optionally output the mesh generated from data point locations as a tetrahedral volume mesh.
- **Modified the `Makefile` in `src/IO_generator`**. Previously, (undocumented) user intervention was required to rebuild the mesh generation libraries with custom I/O formats. Now this is done when one would expect; specifically, running `make` in the directory `src` or in the GRUMMP root directory is sufficient to rebuild the I/O routines and add them to the GRUMMP libraries, then rebuild the executables.
- **Minor code cleanup**, fixing some (but not all) small complaints that HP-UX 10.20 `CC` and `cc` compilers had.
- `configure` is now even **smarter about finding the `flex`/`lex` library**. Since some versions of `flex`/`lex` define `main()` as a macro and some define `yywrap()` as a macro, neither is certain to be in the library. No known cases exist where *both* `main()` and `yywrap()` are macros, so checking for both ensures that the library will be correctly identified if it exists.
- `configure` now accepts the options `--with-c-compiler=NAME` and `--with-cxx-compiler=NAME` to **allow specification of a native C or C++ compiler** to be used if present in preference over `gcc`/`g++`. On most machines this means checking for `cc` and `CC`; for AIX and OSF systems, the correct local names are used. Also, `configure` is smarter about paths when checking for native compilers by full path name.

A.19 From version 0.1.1 to 0.1.2:

- Fixed problem with **internal `snprintf`** not null-terminating strings (caused bad file names).
- Cleaned up **`lex` input file syntax** so that stricter versions of `lex` can parse them correctly.
- Added **I/O files** that were generated with `lex` to source distribution **for systems without a working `lex`**.
- Added (at least partial) **support for DEC/OSF1** machines using native compilers.
- Added **native compiler check for IRIX, OSF1, and AIX** machines so that the native compilers will be used rather than `gcc`/`g++` when the native compilers exist. This check is being added on a machine-by-machine basis, because some (including SunOS 4.1.x) have broken native compilers.
- Added **shared library support for OSF and FreeBSD** machines.

A.20 From version 0.1.0 to 0.1.1:

- If shared libraries can not be built, **static libraries** are used instead, and a message is printed by `configure` requesting information to allow addition of shared library support.
- `configure` is now **smarter about finding the `flex`/`lex` library**. In particular, `/usr/local/lib` is always checked, and a path name can be given to `configure` using `--with-flex-lib-dir=DIRNAME`.
- If `lex` must be used instead of `flex`, **don't request a case-insensitive lexer**. Necessary because `lex -i` fails on many systems.
- Fixed some **`Makefile` problems** that affected IRIX 6.x.
- Added **support for Solaris/gcc** machines.
- **Enabled -s option for `tri`**.
- **Improved length scale generation** for two-dimensional meshes.